Python dataclass

Summary: in this tutorial, you'll learn about the Python dataclass decorator and how to use it effectively.

Introduction to the Python dataclass

Python introduced the dataclass in version 3.7 (PEP 557 (https://www.python.org/dev/peps/pep-0557/)). The dataclass allows you to define classes (https://www.pythontutorial.net/python-oop/python-class/) with less code and more functionality out of the box.

The following defines a regular `Person` class with two instance attributes `name` and `age` :

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

This `Person` class has the `__init__` method that initializes the `name` and `age` attributes.

If you want to have a string representation of the `Person` object, you need to implement the `__str__` (https://www.pythontutorial.net/python-oop/python-__str__/) or `__repr__` (https://www.pythontutorial.net/python-oop/python-__repr__/) method. Also, if you want to compare two instances of the Person class by an attribute, you need to implement the __eq__ (https://www.pythontutorial.net/python-oop/python-__eq__/) method.

However, if you use the dataclass, you'll have all of these features (and even more) without implementing these dunder methods.

To make the Person class a data class, you follow these steps:

First, import the dataclass decorator from the dataclasses module:

```
from dataclasses import dataclass
```

Second, decorate the Person class with the dataclass decorator and declare the attributes:

```
@dataclass
class Person:
    name: str
    age: int
```

In this example, the Person class has two attributes name with the type str and age with the type int . By doing this, the @dataclass decorator implicitly creates the __init__ method like this:

```
def __init__(name: str, age: int)
```

Note that the order of the attributes declared in the class will determine the orders of the parameters in the __init__ method.

And you can create the Person 's object:

```
p1 = Person('John', 25)
```

When printing out the Person 's object, you'll get a readable format:

```
print(p1)
```

Output:

```
Person(name='John', age=25)
```

Also, if you compare two `Person` 's objects with the same attribute value, it'll return `True` . For example:

```
p1 = Person('John', 25)
p2 = Person('John', 25)
print(p1 == p2)
```

Output:

```
True
```

The following discusses other functions that a data class provides.

Default values

When using a regular class, you can define default values for attributes. For example, the following `Person` class has the `iq` parameter with the default value of `100` .

```
class Person:
    def __init__(self, name, age, iq=100):
        self.name = name
        self.age = age
        self.iq = iq
```

To define a default value for an attribute in the dataclass, you assign it to the attribute like this:

```
from dataclasses import dataclass


@dataclass
class Person:
    name: str
    age: int
    iq: int = 100


print(Person('John Doe', 25))
```

Like the parameter rules, the attributes with the default values must appear after the ones without default values. Therefore, the following code will not work:

```
from dataclasses import dataclass


@dataclass
class Person:
    iq: int = 100
    name: str
    age: int
```

Convert to a tuple or a dictionary

The `dataclasses` module has the `astuple()` and `asdict()` functions that convert an instance of the dataclass to a tuple (https://www.pythontutorial.net/python-basics/python-tuples/) and a dictionary (https://www.pythontutorial.net/python-basics/python-dictionary/) . For example:

```
from dataclasses import dataclass, astuple, asdict
```

```python
@dataclass
class Person:
    name: str
    age: int
    iq: int = 100


p = Person('John Doe', 25)

print(astuple(p))
print(asdict(p))
```

Output:

```
('John Doe', 25, 100)
{'name': 'John Doe', 'age': 25, 'iq': 100}
```

Create immutable objects

To create readonly objects from a dataclass, you can set the frozen argument of the dataclass decorator to `True`. For example:

```python
from dataclasses import dataclass, astuple, asdict


@dataclass(frozen=True)
class Person:
    name: str
    age: int
    iq: int = 100
```

If you attempt to change the attributes of the object after it is created, you'll get an error. For example:

```
p = Person('Jane Doe', 25)
p.iq = 120
```

Error:

```
dataclasses.FrozenInstanceError: cannot assign to field 'iq'
```

Customize attribute behaviors

If don't want to initialize an attribute in the __init__ method, you can use the `field()` function from the `dataclasses` module.

The following example defines the `can_vote` attribute that is initialized using the `__init__` method:

```
from dataclasses import dataclass, field


class Person:
    name: str
    age: int
    iq: int = 100
    can_vote: bool = field(init=False)
```

The `field()` function has multiple interesting parameters such as `repr` , `hash` , `compare` , and `metadata` .

If you want to initialize an attribute that depends on the value of another attribute, you can use the `__post_init__` method. As its name implies, Python calls the `__post_init__` method after the `__init__` method.

The following use the `__post_init__` method to initialize the `can_vote` attribute based on the `age` attribute:

```python
from dataclasses import dataclass, field


@dataclass
class Person:
    name: str
    age: int
    iq: int = 100
    can_vote: bool = field(init=False)

    def __post_init__(self):
        print('called __post_init__ method')
        self.can_vote = 18 <= self.age <= 70


p = Person('Jane Doe', 25)
print(p)
```

Output:

```
called the __post_init__ method
Person(name='Jane Doe', age=25, iq=100, can_vote=True)
```

Sort objects

By default, a dataclass implements the `__eq__` method.

To allow different types of comparisons like `__lt__` , `__lte__` , `__gt__` , `__gte__` , you can set the order argument of the `@dataclass` decorator to True:

```python
@dataclass(order=True)
```

By doing this, the dataclass will sort the objects by every field until it finds a value that’s not equal.

In practice, you often want to compare objects by a particular attribute, not all attributes. To do that, you need to define a field called `sort_index` and set its value to the attribute that you want to sort.

For example, suppose you have a list of `Person` 's objects and want to sort them by age:

```
members = [
    Person('John', 25),
    Person('Bob', 35),
    Person('Alice', 30)
]
```

To do that, you need to:

- First, pass the `order=True` parameter to the `@dataclass` decorator.
- Second, define the `sort_index` attribute and set its `init` parameter to `False` .
- Third, set the `sort_index` to the `age` attribute in the `__post_init__` method to sort the `Person` 's object by age.

The following shows the code for sorting `Person` 's objects by age:

```
from dataclasses import dataclass, field


@dataclass(order=True)
class Person:
    sort_index: int = field(init=False, repr=False)

    name: str
    age: int
    iq: int = 100
    can_vote: bool = field(init=False)

    def __post_init__(self):
        self.can_vote = 18 <= self.age <= 70
        # sort by age
        self.sort_index = self.age


members = [
    Person(name='John', age=25),
    Person(name='Bob', age=35),
    Person(name='Alice', age=30)
]

sorted_members = sorted(members)
for member in sorted_members:
    print(f'{member.name}(age={member.age})')
```

Output:

```
John(age=25)
Alice(age=30)
Bob(age=35)
```

Summary

- Use the `@dataclass` decorator from the `dataclasses` module to make a class a dataclass. The dataclass object implements the `__eq__` and `__str__` by default.
- Use the `astuple()` and `asdict()` functions to convert an object of a dataclass to a tuple and dictionary.
- Use `frozen=True` to define a class whose objects are immutable.
- Use `__post_init__` method to initalize attributes that depends on other attributes.
- Use `sort_index` to specify the sort attributes of the dataclass objects.