Contents

Data Classes in Action

Table of Contents

What Are Python Dataclasses?

Dataclasses in Python are a way to simplify the process of creating classes that store data. They automatically generate special methods like `__init__()`, __repr__(), __eq__(), and others, based on the class attributes. Introduced in Python 3.7, dataclasses reduce boilerplate code when you primarily need a class to manage data.

Why Do We Need Dataclasses?

- **Reduces boilerplate**: No need to manually write the __init__(), __repr__(), __eq__() methods.
- **Readability**: Code is cleaner and easier to maintain.
- **Customizability**: Dataclasses are still regular Python classes, so you can add additional methods and functionality.

How to Use Dataclasses?

You use the @dataclass decorator from the dataclasses module to define a dataclass. The decorator automatically generates special methods based on the class attributes.

Syntax of a Modern Python Dataclass

Below is the exhaustive syntax for a Python dataclass covering all related concepts:

```python
from dataclasses import dataclass, field, InitVar
from typing import List, Optional

@dataclass(order=True, frozen=False, slots=True)
class Employee:
    # Basic fields with default values
    id: int
    name: str = "Unknown"

    # Optional fields with default factories (lists, dictionaries, etc.)
    skills: List[str] = field(default_factory=list)

    # Init-only variables (not stored as attributes)
    experience: InitVar[int] = 0

    # Private fields with custom metadata
    _salary: float = field(repr=False, compare=False, metadata={"unit": "USD"})

    # Post-init method for further processing after the object is created
```

```python
    def __post_init__(self, experience: int):
        if experience < 5:
            self._salary = 50000
        else:
            self._salary = 70000

    # Example method
    def give_raise(self, amount: float):
        self._salary += amount

# Usage
emp = Employee(id=1, name="Alice", experience=3)
print(emp)  # Output: Employee(id=1, name='Alice', skills=[])
print(emp._salary)  # Output: 50000

emp.give_raise(5000)
print(emp._salary)  # Output: 55000
```

Explanation of Key Features

Feature	Meaning	Usage
`@dataclass`	The decorator that makes the class a dataclass, auto-generating methods like `__init__()`.	`@dataclass` above the class definition.
`order=True`	Generates comparison methods (`<`, `<=`, `>`, `>=`) for the class.	`@dataclass(order=True)` to enable ordering.
`frozen=False`	Makes the instance immutable (if set to `True`).	`@dataclass(frozen=True)` to make the class immutable.
`slots=True`	Optimizes memory usage by not storing attributes in the usual `__dict__` but in a static structure (Python 3.10+).	`@dataclass(slots=True)` for memory efficiency.
`field()`	Fine-tunes the behavior of individual fields (e.g., default values, whether to include in `repr`, comparison, etc.).	`field(default=..., repr=False, compare=False)` to customize field behavior.
`InitVar`	Defines init-only variables, used during initialization but not stored as instance attributes.	`experience: InitVar[int] = 0` — this variable is only available in the constructor and is not an attribute.

Feature	Meaning	Usage
`__post_init__()`	A special method that runs after the `__init__` method. It's useful when you want to process fields after initialization or work with `InitVar`.	Custom logic such as setting `_salary` based on experience inside `__post_init__`.
`default_factory`	Used to set default values for mutable types like lists or dictionaries.	`skills: List[str] = field(default_factory=list)` to initialize a list.
`metadata`	Allows storing additional metadata for a field, useful for validation or documentation.	`_salary: float = field(metadata={"unit": "USD"})` — metadata is just extra information associated with the field.

Dataclass Example vs. Regular Class Example

Regular Class Example

```
class Employee:
    def __init__(self, id, name="Unknown", skills=None, experience=0):
        self.id = id
        self.name = name
        self.skills = skills if skills is not None else []
        self.experience = experience
        self._salary = 50000 if experience < 5 else 70000

    def give_raise(self, amount):
        self._salary += amount

    def __repr__(self):
        return f"Employee(id={self.id}, name={self.name}, skills={self.skills})"

# Usage
emp = Employee(id=1, name="Alice", experience=3)
print(emp)  # Output: Employee(id=1, name='Alice', skills=[])
print(emp._salary)  # Output: 50000

emp.give_raise(5000)
print(emp._salary)  # Output: 55000
```

Dataclass Example

```
from dataclasses import dataclass, field
```

```python
@dataclass
class Employee:
    id: int
    name: str = "Unknown"
    skills: list = field(default_factory=list)
    experience: int = 0
    _salary: float = field(init=False, repr=False)

    def __post_init__(self):
        self._salary = 50000 if self.experience < 5 else 70000

    def give_raise(self, amount: float):
        self._salary += amount

# Usage
emp = Employee(id=1, name="Alice", experience=3)
print(emp)  # Output: Employee(id=1, name='Alice', skills=[])
print(emp._salary)  # Output: 50000

emp.give_raise(5000)
print(emp._salary)  # Output: 55000
```

Summary Table

Concept	Description	Example
What are Dataclasses?	Simplified way to create classes focused on storing data, reducing boilerplate.	`@dataclass` decorator on a class.
Why Dataclasses?	To automatically generate methods like `__init__()`, `__repr__()`, and `__eq__()`.	Less manual code needed.
Usage	Add `@dataclass` decorator and define class attributes like regular class fields.	`@dataclass` and define fields like `id: int`, `name: str`.
Field	Customizes individual fields with defaults, whether they should be compared, etc.	`field(default_factory=list, repr=False)` for a list field.
post_init	Allows post-initialization processing, useful for logic that depends on other fields.	Initialize `_salary` after setting `experience` in the constructor.

Concept	Description	Example
InitVar	Init-only variables that aren't stored as attributes, available only in `__post_init__()`.	`experience: InitVar[int] = 0` can be passed but isn't stored.
Slots	Optimizes memory usage by reducing overhead from `__dict__`.	`@dataclass(slots=True)` to use memory-efficient attribute storage.
Order	Generates ordering methods (`<`, `>`, `<=`, `>=`).	`@dataclass(order=True)` allows comparisons between instances.
Frozen	Makes the class immutable (frozen instances cannot be modified).	`@dataclass(frozen=True)` prevents any modification of instance attributes.
Default Factories	Assign default values for mutable types such as lists or dictionaries.	`field(default_factory=list)` initializes an empty list if no value is provided.

This summary should give you a comprehensive understanding of Python's dataclasses and how to use them effectively!

Why Do We Need `default_factory`?

In Python, mutable default arguments (like lists, dictionaries, sets, etc.) can lead to unexpected behavior. If a mutable default argument is shared across instances of a class, modifying it in one instance can affect other instances.

To avoid this issue, `default_factory` in Python dataclasses is used to create default values for fields that need mutable data types or for dynamically computed values. It ensures each instance gets a fresh, independent object rather than sharing one.

When to Use `default_factory`?

- **Mutable default values**: For fields that need mutable types (lists, dictionaries), using a factory function ensures that each instance has its own independent copy.
- **Dynamic defaults**: When the default value of a field depends on some logic, and you want that logic to run when the class is instantiated (instead of at the time of definition).
- **Lazy initialization**: When an object or value should only be created at runtime.

How to Use `default_factory`?

You use the `field()` function with the `default_factory` argument, passing a callable (usually a function or class) that generates the default value. Here are some examples:

Examples of `default_factory` Usage

1. Using `default_factory` for Mutable Data Containers (List)

```python
from dataclasses import dataclass, field

@dataclass
class Team:
    members: list = field(default_factory=list)

# Each Team instance gets a new empty list for members
team1 = Team()
team2 = Team()

team1.members.append("Alice")
print(team1.members)  # Output: ['Alice']
print(team2.members)  # Output: [] (Not affected by team1)
```

In this example, without `default_factory`, using a default argument like `members: list = []` would result in both `team1` and `team2` sharing the same list.

---

2. Using `default_factory` with a Function

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

def generate_id():
    return random.randint(1000, 9999)

@dataclass
class Employee:
    id: int = field(default_factory=generate_id)
    name: str = "Unknown"

# Each Employee gets a random ID at instantiation
emp1 = Employee(name="Alice")
emp2 = Employee(name="Bob")
```

```python
print(emp1.id)  # Output: Random ID, e.g., 1203
print(emp2.id)  # Output: Different random ID, e.g., 4537
```

Here, `generate_id()` is a function that provides a dynamic default value, ensuring each employee gets a unique ID.

---

3. Using `default_factory` for Complex Nested Structures

```python
from dataclasses import dataclass, field
from typing import Dict

@dataclass
class Inventory:
    stock: Dict[str, int] = field(default_factory=lambda: {'apple': 0, 'banana': 0})

# Each Inventory instance gets its own dictionary
inv1 = Inventory()
inv2 = Inventory()

inv1.stock['apple'] += 10
print(inv1.stock)  # Output: {'apple': 10, 'banana': 0}
print(inv2.stock)  # Output: {'apple': 0, 'banana': 0} (Not affected by inv1)
```

This example shows how `default_factory` can be used to initialize nested data structures, like a dictionary.

---

4. Using `default_factory` for Lazy Initialization Sometimes, you might want a field to be initialized only when the class is instantiated, which is especially useful for expensive computations.

```python
@dataclass
class DatabaseConnection:
    conn: str = field(default_factory=lambda: "Connected to DB")

@dataclass
class Application:
    db_connection: DatabaseConnection = field(default_factory=DatabaseConnection)

app = Application()
print(app.db_connection.conn)  # Output: Connected to DB
```

In this example, the `DatabaseConnection` object is only created when the `Application` class is instantiated.

Summary Table for `default_factory`

Use Case	Description	Example
Mutable Defaults	Use `default_factory` to avoid shared mutable default arguments.	`field(default_factory=list)` for a list.
Dynamic Default Values	Use it to generate dynamic values at runtime, e.g., random IDs or values based on logic.	`field(default_factory=generate_id)` where `generate_id()` returns a unique value for each instance.
Complex Data Structures	Use it for initializing fields with complex types like dictionaries or other data containers.	`field(default_factory=lambda: {'key': 0})` for initializing a dictionary.
Lazy Initialization	Create expensive or delayed objects only at the time of instantiation.	`field(default_factory=DatabaseConnection)` ensures a connection object is created when needed.

Why Do We Need `__post_init__()`?

In Python dataclasses, `__post_init__()` is a special method that gets called after the dataclass `__init__()` method completes. This is useful when you need to perform some additional logic after the object is initialized, especially when you have fields like `InitVar` that are not stored as attributes or when you need to perform computations that depend on multiple fields.

When to Use `__post_init__()`?

- **Post-constructor initialization**: When additional initialization or validation is required after the default `__init__()` is run.
- **Working with `InitVar`**: You can handle `InitVar` variables (those that are passed during initialization but are not stored as attributes) in `__post_init__()`.
- **Dependent computations**: If certain attributes need to be set or computed based on the values of other fields, `__post_init__()` is a good place to do that.

How to Use `__post_init__()`?

You define a `__post_init__()` method within your dataclass. This method is automatically called right after the `__init__()` method, allowing you to add any logic you want.

Examples of `__post_init__()` Usage

1. Basic Usage of `__post_init__()`

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int
    status: str = "Unknown"

    def __post_init__(self):
        if self.age >= 18:
            self.status = "Adult"
        else:
            self.status = "Minor"

# Usage
person1 = Person(name="Alice", age=20)
person2 = Person(name="Bob", age=15)

print(person1)  # Output: Person(name='Alice', age=20, status='Adult')
print(person2)  # Output: Person(name='Bob', age=15, status='Minor')
```

In this example, `__post_init__()` is used to set the `status` field based on the `age` value after initialization.

---

2. Using `InitVar` with `__post_init__()` `InitVar` allows you to pass values during initialization but not store them as attributes. You can process them in `__post_init__()`.

```
from dataclasses import dataclass, field, InitVar

@dataclass
class Product:
    name: str
    price: float
    discount: InitVar[float] = 0.0  # Passed during init, not stored as an attribute
    final_price: float = field(init=False)

    def __post_init__(self, discount: float):
```

```python
        self.final_price = self.price - (self.price * discount)

# Usage
product = Product(name="Laptop", price=1000, discount=0.1)
print(product)  # Output: Product(name='Laptop', price=1000, final_price=900.0)
```

In this case, the `discount` is an `InitVar` that is passed during initialization but is only used inside `__post_init__()` to compute the `final_price`.

---

3. Handling Field Validation in `__post_init__()` You can use `__post_init__()` to validate fields after initialization.

```python
from dataclasses import dataclass

@dataclass
class BankAccount:
    account_number: str
    balance: float

    def __post_init__(self):
        if self.balance < 0:
            raise ValueError("Balance cannot be negative!")

# Usage
try:
    account = BankAccount(account_number="123ABC", balance=-100)
except ValueError as e:
    print(e)  # Output: Balance cannot be negative!
```

Here, `__post_init__()` checks whether the `balance` is negative and raises an error if it is.

---

4. Using `__post_init__()` for Dependent Initialization When one field's initialization depends on another, `__post_init__()` can help handle that.

```python
from dataclasses import dataclass, field

@dataclass
class Rectangle:
    width: float
    height: float
    area: float = field(init=False)

    def __post_init__(self):
```

```
        self.area = self.width * self.height

# Usage
rect = Rectangle(width=5, height=10)
print(rect.area)  # Output: 50
```

Here, the `area` field is computed based on the `width` and `height` fields during `__post_init__()`.

Summary Table for `__post_init__()`

Use Case	Description	Example
Post-constructor initialization	Allows further processing after `__init__()`, particularly when the class's default constructor isn't enough to initialize all values correctly.	Set `status` based on `age` after the object is initialized.
Working with `InitVar`	Enables processing of variables passed during initialization but not stored as attributes.	Use `discount` to calculate `final_price` without storing `discount`.
Dependent computations	Useful for fields that depend on the values of other fields, especially when computed dynamically.	Compute `area` after initializing `width` and `height`.
Field validation	Allows validation of fields after the object is constructed, ensuring valid values (e.g., checking for negative balances).	Raise an error if the `balance` is negative in a `BankAccount` class.

Let me know if you'd like more detailed examples or have other questions about dataclasses!

1. `init`: Add `__init__()` Method?

What It Is: The `init` parameter controls whether the `__init__()` method should be auto-generated for your dataclass. If set to `True` (the default), the dataclass will generate an `__init__()` method based on the fields defined in the class.

Why We Need It: In most cases, we want dataclasses to auto-generate the `__init__()` method, saving us the trouble of manually writing it. However, there are cases where you may want to manage the initialization yourself (e.g., when more complex logic is required or when using another design pattern).

Example:

```
from dataclasses import dataclass

@dataclass(init=True)
class Car:
    make: str
    model: str

car = Car(make="Toyota", model="Corolla")
print(car)  # Output: Car(make='Toyota', model='Corolla')

# If init=False, you must provide your own constructor.
@dataclass(init=False)
class CarManualInit:
    make: str
    model: str

    def __init__(self, make, model):
        self.make = make
        self.model = model

car_manual = CarManualInit(make="Toyota", model="Corolla")
print(car_manual)  # Output: CarManualInit(make='Toyota', model='Corolla')
```

---

2. `repr`: Add `__repr__()` Method?

What It Is: The `repr` parameter controls whether a `__repr__()` method should be auto-generated for your dataclass. This method returns a string that represents the object in a readable way, useful for debugging or logging.

Why We Need It: Having a good `__repr__()` method makes it easier to inspect and debug objects. However, in cases where you want to hide certain fields or provide a custom representation, you can set `repr=False` or manually define your own `__repr__()` method.

Example:

```
@dataclass(repr=True)
class Employee:
    name: str
    position: str
```

```python
emp = Employee(name="Alice", position="Engineer")
print(emp)  # Output: Employee(name='Alice', position='Engineer')

# Custom repr or hiding field from repr:
@dataclass(repr=False)
class ConfidentialEmployee:
    name: str
    position: str

emp_conf = ConfidentialEmployee(name="Bob", position="Manager")
print(emp_conf)  # Output: <__main__.ConfidentialEmployee object at 0x...>
```

---

3. eq: Add __eq__() Method?

What It Is: The `eq` parameter controls whether the dataclass should auto-generate the `__eq__()` method, which compares two objects for equality. By default, it checks whether the values of all the fields are the same between two instances.

Why We Need It: The `__eq__()` method is useful for checking equality between instances. In certain cases, you may not want equality to depend on all fields, or you may want to implement a custom equality method.

Example:

```python
@dataclass(eq=True)
class Product:
    name: str
    price: float

prod1 = Product(name="Phone", price=699.99)
prod2 = Product(name="Phone", price=699.99)
print(prod1 == prod2)  # Output: True

# eq=False disables the automatic equality check:
@dataclass(eq=False)
class ManualProduct:
    name: str
    price: float

prod_manual = ManualProduct(name="Phone", price=699.99)
prod_manual2 = ManualProduct(name="Phone", price=699.99)
print(prod_manual == prod_manual2)  # Output: False (No equality check)
```

---

4. `order`: Add Ordering Methods?

What It Is: The `order` parameter controls whether comparison methods (`<`, `<=`, `>`, `>=`) are auto-generated. If `order=True`, the dataclass will add ordering methods based on the field values.

Why We Need It: If you want to compare instances of a dataclass (e.g., sorting employees by salary), you can set `order=True` to auto-generate the comparison methods.

Example:

```python
@dataclass(order=True)
class Employee:
    name: str
    salary: float

emp1 = Employee(name="Alice", salary=50000)
emp2 = Employee(name="Bob", salary=60000)

print(emp1 < emp2)  # Output: True (because 50000 < 60000)

# order=False by default, so no ordering methods are generated.
@dataclass(order=False)
class UnorderedEmployee:
    name: str
    salary: float

# This will raise an error if you try to compare instances
# unordered_emp1 < unordered_emp2  -> TypeError: '<' not supported
```

---

5. `unsafe_hash`: Force the Addition of a `__hash__()` Method?

What It Is: The `unsafe_hash` parameter forces the generation of a `__hash__()` method, even when the class is mutable. Normally, mutable classes (which can change their values) should not be hashable because their hash might change when fields change.

Why We Need It: Use `unsafe_hash=True` if you need instances of your dataclass to be hashable (e.g., to be used in sets or as dictionary keys), but be careful—changing a field after hashing can lead to inconsistent results.

Example:

```python
@dataclass(unsafe_hash=True)
class Product:
    name: str
    price: float

prod = Product(name="Laptop", price=1000)
print(hash(prod))  # Hash is generated and can be used as a key in dictionaries

# unsafe_hash=False (default), so normally this would raise an error if you try to hash
@dataclass(unsafe_hash=False)
class UnhashableProduct:
    name: str
    price: float

# unhashable_prod = UnhashableProduct(name="Phone", price=699)
# hash(unhashable_prod) -> TypeError: unhashable type: 'UnhashableProduct'
```

---

6. `frozen`: Make the Instance Immutable?

What It Is: The `frozen` parameter makes a dataclass immutable. If `frozen=True`, the fields cannot be changed after initialization, and attempts to modify them will raise an exception.

Why We Need It: If you need your dataclass to be immutable (like a tuple), use `frozen=True`. This is especially useful when you want instances to be hashable, as immutability guarantees consistent hash values.

Example:

```python
@dataclass(frozen=True)
class ImmutableEmployee:
    name: str
    position: str

emp = ImmutableEmployee(name="Alice", position="Engineer")
# emp.name = "Bob"  # Raises: FrozenInstanceError: cannot assign to field 'name'

# frozen=False (default), allows field modification
@dataclass(frozen=False)
class MutableEmployee:
    name: str
    position: str
```

```python
emp_mutable = MutableEmployee(name="Alice", position="Engineer")
emp_mutable.name = "Bob"  # Works because the instance is mutable
print(emp_mutable.name)  # Output: Bob
```

Summary Table for `@dataclass()` Parameters

Parameter	Description	Default	Example
`init`	Automatically adds an `__init__()` method that initializes the fields.	`True`	`@dataclass(init=False)` if you want to manually define the constructor.
`repr`	Automatically adds a `__repr__()` method for a readable string representation.	`True`	`@dataclass(repr=False)` if you don't want the default string representation.
`eq`	Automatically adds an `__eq__()` method for checking equality between instances.	`True`	`@dataclass(eq=False)` if you want to disable automatic equality checks.
`order`	Automatically adds ordering methods (`<`, `<=`, `>`, `>=`) for comparing instances.	`False`	`@dataclass(order=True)` if you want to enable instance comparison.
`unsafe_hash`	Forces the addition of a `__hash__()` method, even for mutable instances. Use with caution, as it can lead to inconsistent behavior if fields are changed.	`False`	`@dataclass(unsafe_hash=True)` to make instances hashable, even if the class is mutable.
`frozen`	Makes the dataclass immutable (fields cannot be modified after initialization).	`False`	`@dataclass(frozen=True)` if you want to make the instance immutable like a tuple.

This breakdown covers each parameter with explanations and examples, showing how they can customize the behavior of dataclasses. Let me know if you'd like more details!