dc-02

December 1, 2025

[35]:
```
COLORS = {
    'header': '\033[95m',
    'blue': '\033[94m',
    'cyan': '\033[96m',
    'green': '\033[92m',
    'warning': '\033[93m',
    'fail': '\033[91m',
    'endc': '\033[0m',
    'bold': '\033[1m',
    'underline': '\033[4m'
}

def color_text(text: str, color: str) -> str:
    return f"{COLORS.get(color, COLORS['endc'])}{text}{COLORS['endc']}"
```

0.1 Deep Dive into dataclasses

0.1.1 What are dataclasses — and why were they introduced (the historical / problem context)

- Dataclasses were introduced in Python **3.7**, via `PEP 557`. (Python documentation)
- The design goal was to make it much easier to write classes that are "plain data holders" — i.e. classes whose main purpose is to store a set of fields/attributes, without a lot of custom logic. (Python Enhancement Proposals (PEPs))
- Why was this needed? Because before dataclasses, if you wanted a class just to hold data, you had basically three suboptimal options:
    1. Use a plain class and manually write `__init__`, `__repr__`, `__eq__`, etc. — which is tedious, boilerplate-heavy, and error-prone. (realpython.com)
    2. Use a tuple or list — but then you have no named fields, and it's easy to get order wrong or access by index incorrectly. (realpython.com)
    3. Use a dictionary — but then you lose structure, type hints, and are more prone to typos or missing keys. (Medium)

[ ]:
```
# First Method: Plain Class Definition:
from typing import Self

class UserPlain:
```

```python
    def __init__(
            self: Self,
            id: int,
            email: str,
            name: str,
            age: int,
        ) -> None:
        self.id = id
        self.email = email
        self.name = name
        self.age = age

    def __repr__(self: Self) -> str:
        return f"UserPlain(id={self.id!r}, email={self.email!r}, name={self.
↪name!r}, age={self.age!r})"

    # NOTE 01: More correct way to type hint 'other' parameter is 'object', but␣
↪we also can use 'UserPlain' directly.
    # def __eq__(self: Self, other: "UserPlain") -> bool:
    def __eq__(self: Self, other: object) -> bool:
        if not isinstance(other, UserPlain):
            return NotImplemented
        return (
            self.id == other.id and
            self.email == other.email and
            self.name == other.name and
            self.age == other.age
        )
    # NOTE 02: Alternative implementation of __eq__ method:
    # return (self.id, self.email, self.name, self.age) == (other.id, other.
↪email, other.name, other.age)

# Usage Example:
u1 = UserPlain(id=1, email="neo@email.com", name="Neo", age=27)
u2 = UserPlain(id=1, email="neo@email.com", name="Neo", age=27)

print(f"User: {u1}")
print(f"Are u1 and u2 equal? {u1 == u2}")
```

```
User: UserPlain(id=1, email='neo@email.com', name='Neo', age=27)
Are u1 and u2 equal? True
```

Problems / downsides: you have to manually repeat field names across `__init__`, `__repr__`, `__eq__`. As you add more fields, boilerplate grows; risk of typos, forgot fields, outdated repr/eq, etc.

[ ]:
```python
# Second Method: Using Tuple or List as Data Container:
from typing import Tuple, List, Any

# User as TUPLE: id, name, email, age
user_tuple: Tuple[int, str, str, int] = (1, "Neo", "neo@email.com", 30)

# Access by index
user_id = user_tuple[0]
user_name = user_tuple[1]
user_email = user_tuple[2]
user_age = user_tuple[3]

print("User as Tuple:")
print(f"\tUser Tuple: {user_tuple}")
print(f"\tUser Name: {user_name} | User Email: {user_email}")

print()

# User as LIST: id, name, email, age
user_list: List[Any] = [1, "Neo", "neo@email.com", 30]

# Access by index
user_id = user_list[0]
user_name = user_list[1]
user_email = user_list[2]
user_age = user_list[3]

print("User as List:")
print(f"\tUser List: {user_list}")
print(f"\tUser Name: {user_name} | User Email: {user_email}")
```

```
User as Tuple:
        User Tuple: (1, 'Alice', 'alice@example.com', 30)
        User Name: Alice | User Email: alice@example.com

User as List:
        User List: [1, 'Alice', 'alice@example.com', 30]
        User Name: Alice | User Email: alice@example.com
```

Problems / downsides: You have no explicit field names. Someone reading the code doesn't know what `user_tuple[2]` means. Mistakes in ordering lead to subtle bugs. Harder to maintain or extend if you want to add new fields.

---

[28]:
```python
# Third Method: Using Dictionary as Data Container:
from typing import Dict, Any

# User as DICTIONARY: id, name, email, age
```

```python
user_dict: Dict[str, Any] = {
    "id": 1,
    "name": "Neo",
    "email": "neo@mail.com",
    "age": 30,
}

# Access by key
user_id = user_dict["id"]
user_name = user_dict["name"]
user_email = user_dict["email"]
user_age = user_dict["age"]

print("User as Dictionary:")
print(f"\tUser Dict: {user_dict}")
print(f"\tUser Name: {user_name} | User Email: {user_email}")

# Adding or Removing fields dynamically:
user_dict["is_active"] = True
del user_dict["age"]

print()
print("Updated User as Dictionary:")
print(f"\tUser Dict: {user_dict}")
```

```
User as Dictionary:
        User Dict: {'id': 1, 'name': 'Neo', 'email': 'neo@mail.com', 'age': 30}
        User Name: Neo | User Email: neo@mail.com

Updated User as Dictionary:
        User Dict: {'id': 1, 'name': 'Neo', 'email': 'neo@mail.com',
'is_active': True}
```

Problems / downsides: although you have field names, you're missing structural guarantees: any key can be added, removed, misspelled. No attribute-style access (`user.x`), only key lookups. No help from IDEs or type checkers about what keys exist or what type is expected. As data structures grow, dictionaries become harder to manage cleanly.

Also — as many discuss — dictionaries are generic "mapping" types, not "record / schema" definitions. A dict is flexible, but lacks structure, which can lead to inconsistency across code. (Stack Overflow)

- Dataclasses solve this by offering a syntax similar to record/struct definitions: you declare a class, annotate the fields with types, and decorate with `@dataclass`. The machinery generates a sensible `__init__`, `__repr__`, `__eq__`, and other "dunder" methods automatically. (Python documentation)
- This means you get a clean, maintainable, less error-prone way to define data structures — and you avoid repeating `boilerplate code` every time you need a data container. As one

source puts it: you move from "verbose, boilerplate class definitions" to concise, readable "data containers." (Medium)
- Phrased differently: dataclasses give you a middle-ground between "bare dicts / tuples" and "fully behavioral classes": you get the structure and clarity of classes, but without writing all the plumbing manually.

So — historically / conceptually — dataclasses emerged to fill the need for a lightweight, standardized, and type-hint-friendly way to define data containers in Python, reducing boilerplate and mistakes.

---

1 `Quick Recap:` What is a decorator (in Python)

1.0.1 Definition & basic idea:

- In Python, a **decorator** is any *callable* (usually a function) that takes another function or class as input, and returns a modified version of it (or a new object). (realpython.com)
- The decorator syntax — using the `@` symbol — is syntactic sugar. For example:

```
@my_decorator
def f(...) -> ...:
    ...
```

- Because of this, when we apply a decorator to a **class** (not just a function), the decorator receives the class object, can inspect or modify it (adding methods/attributes), and returns a modified class. This is how class decorators work. (Python Enhancement Proposals (PEPs))

1.0.2 What does a decorator let we do, and WHY and WHEN use it:

Decorators provide a way to **separate concerns**: We can add or modify functionality (like logging, validation, automatic method generation, caching, access control, etc.) *without modifying the original function/class definition.* (realpython.com)

Some common uses of decorators:

- For functions: logging, timing, caching/memoization, authorization, input validation, etc. (GeeksforGeeks)
- For classes: modifying or enhancing class behavior — for instance, automatically generating methods, injecting extra functionality, metaprogramming patterns. This is what `@dataclass` does. (Python Enhancement Proposals (PEPs))

1.0.3 Why `@dataclass` is implemented as a decorator (and what it does behind the scenes)

- `@dataclass` is a **class decorator**. When you apply it to a class, it inspects the class definition: specifically, the **type annotations** of class variables to treat as "fields." (Python Enhancement Proposals (PEPs))
- Based on those fields, the decorator automatically generates several "dunder" (special) methods for the class: e.g., `__init__`, `__repr__`, `__eq__`, (optionally `__lt__`, ordering methods), maybe `__hash__`, depending on configuration. (Python documentation)

- The result: instead of writing all that boilerplate methods manually, you write only the **field definitions** (with types), decorate with `@dataclass`, and get a fully-functional "data container" class. This keeps code concise, maintainable and less error-prone. (GeeksforGeeks)

1.0.4 A Minimal Example of a Decorator (function version)

To illustrate the decorator pattern — for a function — here is a simple example:

[43]:
```
from typing import Callable, Any

def my_decorator(func: Callable[..., Any]) -> Callable[..., Any]:
    def wrapper(*args: Any, **kwargs: Any) -> Any:
        print(f"{color_text('Before calling', 'cyan')}: {color_text(str(func.
 ↪__name__), 'warning')}")
        result = func(*args, **kwargs)
        print(f"{color_text('After calling', 'cyan')}: {color_text(str(func.
 ↪__name__), 'warning')}")
        return result
    return wrapper

@my_decorator
def say_hello(name: str) -> None:
    print(f"Hello, {name}!")

say_hello("Alice")
```

```
Before calling: say_hello
Hello, Alice!
After calling: say_hello
```

- Here, `my_decorator` takes the function `say_hello`, wraps its behavior with extra printing, and returns the "new" function `wrapper`.
- The `@my_decorator` syntax is just short for `say_hello = my_decorator(say_hello)`. (GeeksforGeeks)

1.0.5 Why it's useful to briefly explain/understand decorators before showing `@dataclass`

- Explaining decorators helps remove the "magic" feeling around `@dataclass`. Since often people see `@dataclass` and treat it as "special syntax" — but understanding it as a decorator demystifies what's going on: under the hood, Python is just modifying the class definition.
- It builds `meta-programming` awareness: we learn that in Python we can write "code that writes code" (or modifies definitions, like we did in the HomeWork, for example), which is powerful and pervasive (in libraries, frameworks, etc.).
- It helps draw parallels — in future when we learn other decorator-based libraries (or even function decorators), we can better understand the mechanism.
- It highlights clear trade-offs: decorator-based automation vs manual explicit code — which connects well to the theme of "boilerplate reduction vs explicitness / clarity".

1.1 Key Features & Capabilities of Dataclasses

When we use `@dataclass`, we get automatically generated methods and features, based on the class annotations. According to the official docs: (Python documentation)

Main benefits:

- Automatic `__init__()` — no need to write constructors manually. The parameters correspond to the annotated fields. (Python documentation)
- Automatic `__repr__()` — nice human-readable representation of instances, useful for debugging. (Python documentation)
- Automatic `__eq__()` (and when configured, ordering methods) — means structural equality: two instances compare equal if all their fields are equal. (Python Enhancement Proposals (PEPs))
- Support for default field values, including "default factories" (for mutable defaults). (hamatti.org)
- Ability to make classes immutable (i.e. "frozen dataclasses") via `@dataclass(frozen=True)` — useful when you want hashable / unmodifiable value objects. (hamatti.org)
- Works seamlessly with Python's static type hints (`PEP-526`): so we define fields with types, and dataclasses are fully compatible with type-checking tools. (Python Enhancement Proposals (PEPs))
- Convenience conversion: using helpers like `dataclasses.asdict()` and `dataclasses.astuple()` we can convert instances to dictionaries or tuples — useful for serialization or transformations. (realpython.com)

In short: dataclasses give us a simple, clean, standard way to create data-holding classes with minimal boilerplate — but still with types, defaults, immutability options, and convenience methods.

---

2 Let's Start Coding with Dataclasses!

Let's dive into some practical examples of how to use dataclasses in Python. Using the same examples from the previous cells in the notebook, we will now implement them using dataclasses to see how they simplify our code and reduce boilerplate.

[ ]:
```python
from dataclasses import dataclass

@dataclass
class UserDataclass:
    id: int
    email: str
    name: str
    age: int

# Usage Example:
u1 = UserDataclass(id=1, email="neo@email.com", name="Neo", age=27)
u2 = UserDataclass(id=1, email="neo@email.com", name="Neo", age=27)
```

```
print(f"User: {u1}")
print(f"Are u1 and u2 equal? {u1 == u2}")
```

```
User: UserDataclass(id=1, email='neo@email.com', name='Neo', age=27)
Are u1 and u2 equal? True
```

Much easier, don't you think?

Now, let's, step by step, explore more advanced features of dataclasses, such as default values, immutability, and custom methods.

1. Imagine that we have Users, and this User has an Address related to it. Let's take a look at how we can implement this using dataclasses.

[54]:
```
from dataclasses import dataclass

@dataclass
class Address:
    street: str
    city: str
    zip_code: str

@dataclass
class User:
    id: int
    email: str
    name: str
    age: int
    address: Address

# Usage Example:
address = Address(
    street="Baker Street, 221B",
    city="London",
    zip_code="W1U 6SG",
)
user = User(
    id=1,
    email="sherlock@email.com",
    name="Sherlock Holmes",
    age=40,
    address=address,
)

print(f"User with Address: \n{user}")
```

```
User with Address:
User(id=1, email='sherlock@email.com', name='Sherlock Holmes', age=40,
address=Address(street='Baker Street, 221B', city='London', zip_code='W1U 6SG'))
```

2. Let's imagine that we want to create a dataclass for a Product, which has a name, price, and description, and let's create a Inventory class that holds a list of Products and that provide methods to add, remove, and list products. Let's see how we can implement this using dataclasses.

[ ]:
```python
from typing import Self, List
from dataclasses import dataclass


@dataclass
class Product:
    name: str
    price: float
    description: str


@dataclass
class Inventory:
    products: List[Product]

    def add_product(self: Self, product: Product) -> None:
        self.products.append(product)

    def add_products(self: Self, products: List[Product]) -> None:
        self.products.extend(products)

    def remove_product(self: Self, product: Product) -> None:
        self.products.remove(product)

    def clean_inventory(self: Self) -> None:
        self.products.clear()

    def list_products(self: Self) -> List[Product]:
        return self.products

# Usage Example:
inventory = Inventory(products=[])
print("Initial Inventory:", inventory.list_products())

# 1. Adding products one by one:
product1 = Product(name="Laptop", price=999.99, description="A high-performance␣
 ↪laptop.")
product2 = Product(name="Smartphone", price=499.99, description="A latest model␣
 ↪smartphone.")

inventory.add_product(product1)
inventory.add_product(product2)
```

```
print("\nInventory after adding products:", inventory.list_products())

# 2. Clearing the inventory:
inventory.clean_inventory()
print("\nInventory after cleaning:", inventory.list_products())

# 3. Adding multiple products at once:
product3 = Product(name="Tablet", price=299.99, description="A lightweight␣
 ↪tablet.")
product4 = Product(name="Headphones", price=199.99,␣
 ↪description="Noise-cancelling headphones.")
inventory.add_products([product3, product4])
print("\nInventory after adding multiple products:", inventory.list_products())

# 4. Removing a product:
inventory.remove_product(product3)
print("\nInventory after removing a product:", inventory.list_products())
```

```
Initial Inventory: []

Inventory after adding products: [Product(name='Laptop', price=999.99,
description='A high-performance laptop.'), Product(name='Smartphone',
price=499.99, description='A latest model smartphone.')]

Inventory after cleaning: []

Inventory after adding multiple products: [Product(name='Tablet', price=299.99,
description='A lightweight tablet.'), Product(name='Headphones', price=199.99,
description='Noise-cancelling headphones.')]

Inventory after removing a product: [Product(name='Headphones', price=199.99,
description='Noise-cancelling headphones.')]
```

2.1. Let's add some interactive print statements to see the state of the inventory after each operation.

[67]:
```
from typing import Self, List, Callable
from dataclasses import dataclass


def inventory_logging_decorator(func: Callable[..., None]) -> Callable[...,␣
 ↪None]:
    def wrapper(*args: Any, **kwargs: Any) -> None:
        print(f"{color_text('Logging action', 'green')}: {color_text(str(func.
 ↪__name__), 'blue')}")
        return func(*args, **kwargs)
    return wrapper
```

```python
@dataclass
class Product:
    name: str
    price: float
    description: str


@dataclass
class Inventory:
    products: List[Product]

    @inventory_logging_decorator
    def add_product(self: Self, product: Product) -> None:
        self.products.append(product)

    @inventory_logging_decorator
    def add_products(self: Self, products: List[Product]) -> None:
        self.products.extend(products)

    @inventory_logging_decorator
    def remove_product(self: Self, product: Product) -> None:
        self.products.remove(product)

    @inventory_logging_decorator
    def clean_inventory(self: Self) -> None:
        self.products.clear()

    @inventory_logging_decorator
    def list_products(self: Self) -> List[Product]:
        return self.products

# Usage Example:
inventory = Inventory(products=[])
print(f"Initial Inventory: {inventory.list_products()}\n")

# 1. Adding products one by one:
product1 = Product(name="Laptop", price=999.99, description="A high-performance␣
 ↪laptop.")
product2 = Product(name="Smartphone", price=499.99, description="A latest model␣
 ↪smartphone.")

inventory.add_product(product1)
inventory.add_product(product2)
print(f"Inventory after adding products: {inventory.list_products()}\n")

# 2. Clearing the inventory:
```

```
inventory.clean_inventory()
print(f"Inventory after cleaning: {inventory.list_products()}\n")

# 3. Adding multiple products at once:
product3 = Product(name="Tablet", price=299.99, description="A lightweight␣
 ↪tablet.")
product4 = Product(name="Headphones", price=199.99,␣
 ↪description="Noise-cancelling headphones.")
inventory.add_products([product3, product4])
print(f"Inventory after adding multiple products: {inventory.
 ↪list_products()}\n")

# 4. Removing a product:
inventory.remove_product(product3)
print(f"Inventory after removing a product: {inventory.list_products()}\n")
```

```
Logging action: list_products
Initial Inventory: []

Logging action: add_product
Logging action: add_product
Logging action: list_products
Inventory after adding products: [Product(name='Laptop', price=999.99,
description='A high-performance laptop.'), Product(name='Smartphone',
price=499.99, description='A latest model smartphone.')]

Logging action: clean_inventory
Logging action: list_products
Inventory after cleaning: []

Logging action: add_products
Logging action: list_products
Inventory after adding multiple products: [Product(name='Tablet', price=299.99,
description='A lightweight tablet.'), Product(name='Headphones', price=199.99,
description='Noise-cancelling headphones.')]

Logging action: remove_product
Logging action: list_products
Inventory after removing a product: [Product(name='Headphones', price=199.99,
description='Noise-cancelling headphones.')]
```

3. **Default values & mutable-safe defaults**:

- Each gets its own members list, so no accidental sharing of mutable defaults.

[ ]:
```
from dataclasses import dataclass, field
from typing import List
```

```
@dataclass
class SuperHero:
    name: str

@dataclass
class Team:
    name: str
    members: List[SuperHero] = field(default_factory=list)

team1 = Team("Avengers")
hero1 = SuperHero(name="Iron Man")
hero2 = SuperHero(name="Captain America")

team1.members.append([hero1, hero2])

team2 = Team("Justice League")
print(team1)
print(team2)
```

```
Team(name='Avengers', members=[[SuperHero(name='Iron Man'),
SuperHero(name='Captain America')]])
Team(name='Justice League', members=[])
```

4. **Frozen / immutable dataclass (value object)**:

- Trying to modify an attribute raises a `FrozenInstanceError`.

[84]:
```
from dataclasses import dataclass

@dataclass(frozen=True)
class Point:
    x: float
    y: float

p = Point(1.0, 2.0)
print(p)

try:
    p.x = 3.0
except Exception as e:
    print(f"{color_text('Error', 'warning')}: {color_text(str(e), 'fail')}")
```

```
Point(x=1.0, y=2.0)
Error: cannot assign to field 'x'
```

5. **Keyword-only constructor parameters**:

- Enforces keyword arguments in the constructor for better clarity.

[89]:
```python
from dataclasses import dataclass

@dataclass(kw_only=True)
class Config:
    host: str
    port: int

# Must call with keywords:
cfg = Config(host="localhost", port=8000)
print(f"Configuration Loaded: {color_text(str(cfg), 'green')}")

try:
    cfg2 = Config("localhost", 8000)
except TypeError as e:
    print(f"{color_text('Error', 'warning')}: {color_text(str(e), 'fail')}")
```

```
Configuration Loaded: Config(host='localhost', port=8000)
Error: Config.__init__() takes 1 positional argument but 3 were
given
```

6. **Ordering / sortable objects**:

- Automatically generated comparison methods allow sorting based on field values.

[91]:
```python
from dataclasses import dataclass

@dataclass(order=True)
class Version:
    major: int
    minor: int
    patch: int

v1 = Version(1, 2, 0)
v2 = Version(1, 10, 0)
print(f"Comparison Result: {color_text(str(v1 < v2), 'green')}")
```

```
Comparison Result: True
```

7. Using `slots=True` to reduce memory overhead / disallow dynamic attributes:

- Using `slots=True` in dataclasses reduces memory usage by preventing the creation of `__dict__` for each instance, and disallows adding new attributes dynamically.

[97]:
```python
from dataclasses import dataclass

@dataclass(slots=True)
class User:
    id: int
    name: str
```

```python
# 1. Creating an instance of User:
u = User(1, "Neo")
print(f"User Created: {color_text(str(u), 'green')}")

# 2. Trying to modify an EXISTING attribute:
try:
    u.id = 5
    print(f"User Created: {color_text(str(u), 'green')}")
except Exception as e:
    print(f"{color_text('Error', 'warning')}: {color_text(str(e), 'fail')}")

# 3. Trying to add a NEW attribute dynamically:
try:
    u.new_attr = "hello"
except Exception as e:
    print(f"{color_text('Error', 'warning')}: {color_text(str(e), 'fail')}")
```

```
User Created: User(id=1, name='Neo')
User Created: User(id=5, name='Neo')
Error: 'User' object has no attribute 'new_attr'
```

8. **Mixed control**: skip some fields in init, compute them in `__post_init__`:

- We can exclude certain fields from being initialized via the constructor and instead compute them in the `__post_init__` method, allowing for more complex initialization logic.

[99]:
```python
from typing import Self
from dataclasses import dataclass, field


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

    def __post_init__(self: Self) -> None:
        self.area = self.width * self.height

r = Rectangle(width=3, height=4)
print(f"Area of Rectangle: {color_text(str(r.area), 'green')}")
```

```
Area of Rectangle: 12
```

- **REAL USE CASE**: Building an **API** that need to load ***URL***, ***Passwords*** or classes from `.env` files or external sources. We can use frozen dataclasses to represent configuration objects that should not be modified after creation, ensuring the integrity of your configuration data throughout the application's lifecycle. Also, we can Load each configuration parameter separately from the environment, providing default values or validation as needed, and once loaded, verify each parameter before using it in the application, and only then (iin some cases

like MongoDB URIs) build the final connection string or object.

9. Turning dataclass into tuple-like objects or dictionary-like objects:

- Using `dataclasses.astuple()` and `dataclasses.asdict()` to convert dataclass instances into tuples or dictionaries for easy serialization or manipulation.

[ ]:
```python
from dataclasses import dataclass, astuple, asdict


@dataclass
class User:
    id: int
    name: str


user = User(id=1, name="Neo")
user_tuple = astuple(user)
user_dict = asdict(user)


# Usage Example:
print(f"User as Tuple: {color_text(str(user_tuple), 'green')}")
print(f"User as Dict : {color_text(str(user_dict), 'green')}")
print(f"User as Dataclass: {color_text(str(user), 'green')}")
```

```
User as Tuple: (1, 'Neo')
User as Dict : {'id': 1, 'name': 'Neo'}
User as Dataclass: User(id=1, name='Neo')
```

2.1 When `Dataclasses` are a "*Good*" and when they are "*Bad*" to use:

And what do we mean by that? Like any tool or pattern, dataclasses are not a one-size-fits-all solution. There are scenarios where they shine, and others where they may not be the best fit. So, "Good" and "Bad" here refer to when dataclasses are an appropriate choice versus when they might not be ideal.

When "*Good*":

- You need simple data containers: configurations, DTOs (`data transfer objects`), domain objects that only carry data.
- You want to use type hints, clean syntax, minimal boilerplate, readability.
- You want structural equality, default values, optionally immutability (value objects).
- You want a standard-library solution (no external dependencies).

When maybe *not* ideal:

- If you need validation / parsing / type coercion (e.g. from untrusted input), dataclasses do **not** provide that out of the box. (Python Enhancement Proposals (PEPs))
- If you need advanced behavior beyond "just data" — complex invariants, validation, business logic in initialization — maybe a regular class or a more specialized library is better. Many

argue that mixing heavy logic with dataclasses breaks the "data container" intent. (Redowan's Reflections)

- If you need compatibility with external schemas, JSON schema generation, serialization/deserialization pipelines, nested validation — dataclasses by themselves don't supply those.
- If performance is *extremely* critical (lots of object creation, serialization) and default dataclasses are used with no optimization (e.g. without `slots`) — there is some overhead compared with bare tuples or minimal custom classes. (Redowan's Reflections)

2.2 Summary:

- We explored the historical context and motivation behind dataclasses in Python.
- We learn a little bit about `dataclass` and how it works and how we can use it.

[ ]: