Dataclasses in Python

Introduction

In object oriented programming, many classes exist mainly to **store data**.

For example, a `Resource` object might contain:

- title
- category
- URL
- difficulty
- published status

A traditional Python class can represent this data:

```
class Resource:

    def __init__(self, title, category, url, difficulty):
        self.title = title
        self.category = category
        self.url = url
        self.difficulty = difficulty
```

This works, but notice how much code is required just to assign values.

Python provides **dataclasses** to make these data-focused classes shorter, clearer, and easier to maintain.

---

1. What Is a Dataclass?

A **dataclass** is a Python class designed primarily to store data.

Python's `dataclasses` module can automatically generate common methods such as:

- `__init__()`
- `__repr__()`
- `__eq__()`

depending on how the dataclass is configured.

Basic example:

```
from dataclasses import dataclass


@dataclass
class Resource:
    title: str
    category: str
    url: str
```

Now we can create an object:

```
resource = Resource(
    "Python Basics",
    "Python",
    "/resources/python-basics"
)
```

We automatically get a constructor.

---

2. Why Were Dataclasses Introduced?

Without a dataclass:

```
class Resource:

    def __init__(self, title, category, url):
        self.title = title
        self.category = category
        self.url = url
```

With a dataclass:

```
@dataclass
class Resource:
    title: str
    category: str
    url: str
```

The second version communicates the purpose of the class much more clearly:

> This class mainly represents a piece of data.

---

3. Importing `dataclass`

The `dataclass` decorator comes from Python's standard library:

```
from dataclasses import dataclass
```

Then use:

```
@dataclass
class Resource:
    title: str
    category: str
```

The `@dataclass` decorator tells Python to generate useful class behavior automatically.

---

4. Fields

The variables declared inside a dataclass are called **fields**.

Example:

```
@dataclass
class Resource:
    title: str
    category: str
    difficulty: str
```

Here there are three fields:

```
title
category
difficulty
```

Each field has a type annotation.

```
title: str
```

means:

> `title` is expected to contain a string.

---

5. Type Annotations

Dataclasses rely heavily on type annotations.

Example:

```
@dataclass
class Student:
    name: str
    age: int
    grade: float
```

Create an object:

```
student = Student("Hafsa", 25, 3.8)
```

The annotations make the structure easier to understand.

However, an important point:

> Python does not automatically enforce these type annotations at runtime.

For example:

```
student = Student("Hafsa", "twenty five", "A")
```

Python will generally allow this.

Type annotations primarily provide information to:

- developers
- IDEs
- linters
- type checkers

---

6. Automatic `__init__()`

One of the biggest benefits of dataclasses is automatic constructor generation.

Traditional class:

```
class User:

    def __init__(self, name, email):
        self.name = name
        self.email = email
```

Dataclass:

```
@dataclass
class User:
    name: str
    email: str
```

Python effectively creates the constructor for you.

You can then write:

```
user = User("Hafsa", "hafsa@example.com")
```

---

7. Automatic `__repr__()`

Dataclasses also generate a useful representation of objects.

Example:

```
@dataclass
class Resource:
    title: str
    category: str


resource = Resource("Python Basics", "Python")

print(resource)
```

You get something similar to:

```
Resource(title='Python Basics', category='Python')
```

Without implementing `__repr__()` yourself.

This is particularly useful while debugging.

---

8. Automatic Equality

Dataclasses can also generate `__eq__()`.

Example:

```
@dataclass
class Resource:
    title: str
    category: str
```

Create two objects:

```
resource1 = Resource("Python Basics", "Python")
resource2 = Resource("Python Basics", "Python")
```

Then:

```
print(resource1 == resource2)
```

Output:

```
True
```

The comparison is based on their fields.

This differs from normal object identity comparison.

---

9. Dataclass vs Normal Class

Traditional class:

```
class Resource:

    def __init__(self, title, category):
        self.title = title
        self.category = category

    def __repr__(self):
        return f"Resource(title={self.title!r}, category={self.category!r})"

    def __eq__(self, other):
        if not isinstance(other, Resource):
            return NotImplemented

        return (
            self.title == other.title
            and self.category == other.category
        )
```

Dataclass:

```
@dataclass
class Resource:
    title: str
    category: str
```

The dataclass eliminates a lot of repetitive code.

---

10. Dataclasses Follow the Principle of Less Boilerplate

Boilerplate means repetitive code that appears again and again.

For data-focused classes, code such as:

```
self.name = name
self.email = email
self.age = age
self.role = role
```

can become repetitive.

Dataclasses reduce this repetition.

Instead:

```
@dataclass
class User:
    name: str
    email: str
    age: int
    role: str
```

The class structure becomes easier to read.

---

11. Default Values

Dataclass fields can have default values.

```
@dataclass
class Resource:
    title: str
    category: str = "General"
```

Now:

```
resource = Resource("Python Basics")
```

The category becomes:

```
General
```

You can also override it:

```
resource = Resource("Python Basics", "Python")
```

---

12. Field Ordering Rules

Dataclass fields follow constructor ordering rules.

This works:

```
@dataclass
class User:
    name: str
    age: int = 18
```

But this is problematic:

```
@dataclass
class User:
    age: int = 18
    name: str
```

because a field with a default value comes before a required field.

The same general rule applies as with normal Python function parameters:

> Required fields should come before fields with defaults.

Correct:

```
@dataclass
class User:
    name: str
    age: int = 18
    role: str = "student"
```

---

13. Keyword Arguments

Dataclass objects can be created using keyword arguments.

```
@dataclass
class Resource:
    title: str
    category: str
    difficulty: str
```

Then:

```
resource = Resource(
    title="Python OOP",
    category="Python",
    difficulty="Intermediate"
)
```

This can make object creation easier to read.

---

14. field()

For more control over a dataclass field, Python provides `field()`.

Import it:

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

Example:

```
@dataclass
class Resource:
    title: str
    tags: list[str] = field(default_factory=list)
```

Now:

```
resource = Resource("Python Basics")

print(resource.tags)
```

Output:

```
[]
```

---

15. Why default_factory Matters

This is especially important with mutable values such as:

- lists

- dictionaries
- sets

Avoid:

```
@dataclass
class Resource:
    tags: list[str] = []
```

Instead:

```
@dataclass
class Resource:
    tags: list[str] = field(default_factory=list)
```

`default_factory=list` creates a **new list for each object**.

Example:

```
resource1 = Resource("Python")
resource2 = Resource("JavaScript")

resource1.tags.append("programming")

print(resource1.tags)
print(resource2.tags)
```

Output:

```
['programming']
[]
```

Each object gets its own list.

---

16. `default_factory` With Dictionaries

The same idea works with dictionaries:

```
@dataclass
class User:
    name: str
    settings: dict = field(default_factory=dict)
```

Now every user gets an independent dictionary.

```
user1 = User("Hafsa")
user2 = User("Asim")

user1.settings["theme"] = "dark"

print(user1.settings)
print(user2.settings)
```

Output:

```
{'theme': 'dark'}
{}
```

---

17. `field()` Options

`field()` provides several options.

For example:

```
field(
    default=...,
    default_factory=...,
    init=...,
    repr=...,
    compare=...
)
```

These control how the field behaves in generated methods.

---

18. Excluding a Field From `__init__()`

Use:

```
init=False
```

Example:

```
@dataclass
class Resource:
    title: str
    slug: str = field(init=False)
```

Now the constructor does not expect `slug`.

You can calculate it later.

---

19. Calculated Fields

Example:

```
@dataclass
class Resource:
    title: str
    slug: str = field(init=False)

    def __post_init__(self):
        self.slug = self.title.lower().replace(" ", "-")
```

Now:

```
resource = Resource("Python Basics")

print(resource.slug)
```

Output:

```
python-basics
```

This is useful when a field depends on other fields.

---

20. `__post_init__()`

Dataclasses automatically call `__post_init__()` after the generated `__init__()` finishes.

Example:

```
@dataclass
class User:
    name: str
```

```
    age: int

    def __post_init__(self):
        print("User created")
```

Then:

```
user = User("Hafsa", 25)
```

Output:

```
User created
```

The method is useful for:

- validation
- calculated values
- normalization
- additional initialization

---

21. Validation With `__post_init__()`

Example:

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

    def __post_init__(self):
        if self.age < 0:
```

```
            raise ValueError("Age cannot be negative")
```

Now:

```
user = User("Hafsa", -5)
```

raises an error.

The dataclass itself does not automatically validate the type or value.

You implement validation when needed.

---

22. Normalizing Data

`__post_init__()` can also normalize input.

```
@dataclass
class User:
    name: str
    email: str

    def __post_init__(self):
        self.name = self.name.strip()
        self.email = self.email.strip().lower()
```

Now:

```
user = User(
    "  Hafsa  ",
    "  HAFSA@EXAMPLE.COM "
)
```

The stored values become:

```
Hafsa
hafsa@example.com
```

---

23. Frozen Dataclasses

Sometimes you want objects whose fields cannot be changed after creation.

Use:

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

Create:

```
point = Point(10, 20)
```

Trying:

```
point.x = 50
```

raises an error.

A frozen dataclass provides an immutable-style object.

---

24. Frozen Does Not Mean Deep Immutability

Consider:

```
@dataclass(frozen=True)
class User:
    name: str
    tags: list[str]
```

You cannot replace:

```
user.name = "Asim"
```

but a mutable object inside the dataclass may still be modified:

```
user.tags.append("Python")
```

So:

> `frozen=True` prevents normal attribute assignment; it does not automatically make every nested object immutable.

---

25. Ordering

Dataclasses can optionally generate comparison methods.

Example:

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

Now comparison operations such as:

`product1 < product2`

can be supported based on the fields used for comparison.

Use ordering deliberately.

Do not enable it simply because it is available.

---

26. Comparing Specific Fields

You can control whether a field participates in comparison.

```
@dataclass(order=True)
class Product:
    price: float
    name: str = field(compare=False)
```

Now `price` participates in generated comparisons while `name` does not.

---

27. Excluding Fields From `repr`

Sometimes a field should not appear when the object is printed.

Use:

`repr=False`

Example:

```
@dataclass
class User:
    username: str
    password: str = field(repr=False)
```

Then:

```
user = User("hafsa", "secret")

print(user)
```

The generated representation will omit the password.

This is useful for avoiding accidental exposure in debugging output.

However:

> `repr=False` is not a security mechanism.

The password still exists in the object.

---

28. Excluding Fields From Equality

Use:

```
compare=False
```

Example:

```
@dataclass
class User:
    username: str
    last_login: str = field(compare=False)
```

Now two users with the same username can compare equal even if their `last_login` values differ.

Use this only when the field genuinely should not define object equality.

---

29. `slots=True`

Modern Python versions allow:

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

This creates a dataclass using slots.

Slots can:

- restrict arbitrary new attributes
- reduce per-instance memory usage in some situations
- provide a more constrained object layout

For example:

```
user = User("Hafsa", 25)
```

Then adding an arbitrary new attribute such as:

```
user.country = "Pakistan"
```

is not allowed with the normal slotted behavior.

Use `slots=True` when its constraints and benefits fit your application.

---

30. `kw_only=True`

Dataclasses can require fields to be passed as keyword arguments.

Example:

```
@dataclass(kw_only=True)
class Resource:
    title: str
    category: str
```

Now use:

```
resource = Resource(
    title="Python",
    category="Programming"
)
```

Instead of positional arguments.

This can make APIs safer and easier to read when a class has many fields.

---

31. Dataclass Inheritance

Dataclasses can inherit from other dataclasses.

Example:

```
@dataclass
class Resource:
    title: str


@dataclass
class PDF(Resource):
    pages: int
```

Now:

```
pdf = PDF("Python Basics", 30)
```

The child receives the parent's field.

Dataclasses can therefore work with inheritance when the relationship is appropriate.

---

32. Dataclass Inheritance and Defaults

Default ordering rules still matter when inheritance is involved.

For example, a parent field without a default and a child field with a default generally works naturally:

```
@dataclass
```

```
class Resource:
    title: str


@dataclass
class PDF(Resource):
    pages: int = 10
```

But introducing required fields after inherited default fields can create constructor ordering problems.

When dataclass inheritance becomes complicated, consider whether composition or a different design would be clearer.

---

33. Dataclasses Can Have Methods

A dataclass is still a normal Python class.

It can contain methods.

Example:

```
@dataclass
class Resource:
    title: str
    category: str

    def describe(self):
        return f"{self.title} ({self.category})"
```

Then:

```
resource = Resource("Python Basics", "Python")
```

```
print(resource.describe())
```

Output:

```
Python Basics (Python)
```

Dataclasses are not limited to passive data.

---

34. Dataclass vs Named Tuple

A dataclass and a named tuple can both represent structured data, but they have different purposes.

Dataclass	**Named Tuple**
Mutable by default	Immutable
Can have methods	Can have methods
Supports many configuration options	More lightweight structure
Can use inheritance	Tuple-based
Can contain mutable fields	Tuple itself is immutable
Better suited to rich data objects	Useful for tuple-like records

Choose based on the behavior your data model needs.

---

35. Dataclass vs Dictionary

A dictionary can also store structured information:

```
resource = {
    "title": "Python Basics",
    "category": "Python"
}
```

This is flexible, but it does not clearly define the expected structure.

A dataclass:

```
@dataclass
class Resource:
    title: str
    category: str
```

makes the structure explicit.

Use dictionaries when:

- the structure is highly dynamic
- keys vary
- you are working with arbitrary data

Use dataclasses when:

- the structure is known

- fields have clear meanings
- you want a reusable data model
- type annotations are useful

---

36. Dataclass vs Normal Class

Dataclass	Normal Class
Less boilerplate	More manual code
Automatically generates common methods	You implement them yourself
Excellent for data models	Better when behavior dominates
Clear field declaration	More control over initialization
Easy equality and representation	Full manual control
Can still contain methods	Can contain methods

A dataclass is not a replacement for every class.

---

37. When Should You Use Dataclasses?

Dataclasses are useful for:

- configuration objects
- API response models
- application settings
- user records
- database-like records
- coordinates
- products
- resources
- messages
- events
- DTOs
- structured application data

For example:

```
@dataclass
class Product:
    name: str
    price: float
    category: str
```

This is a natural dataclass use case.

---

38. When Should You Avoid Dataclasses?

A normal class may be better when:

- behavior is much more important than stored data
- object creation requires complicated custom logic
- you need highly customized attribute management
- the class represents a complex abstraction rather than a data record
- automatic equality or representation would be misleading

Do not use a dataclass simply because the class contains attributes.

Ask:

> Is this primarily a data model?

If yes, a dataclass may be a good fit.

---

39. Real haas.dev Example

Imagine a haas.dev resource catalog.

Each resource could have:

- title
- category
- description
- URL
- difficulty
- tags

A traditional class could become verbose.

A dataclass is a natural fit:

```
from dataclasses import dataclass, field


@dataclass
class Resource:
    title: str
    category: str
    description: str
    url: str
    difficulty: str
    tags: list[str] = field(default_factory=list)
```

Create resources:

```
python_resource = Resource(
    title="Python Basics",
    category="Python",
    description="Learn Python fundamentals",
    url="/resources/python-basics",
    difficulty="Beginner",
    tags=["python", "programming"]
)
```

Now the resource is easy to store and work with.

---

40. Adding Behavior to the haas.dev Resource

A dataclass can still contain methods.

```
@dataclass
class Resource:
    title: str
    category: str
    difficulty: str
    tags: list[str] = field(default_factory=list)

    def add_tag(self, tag):
        if tag not in self.tags:
            self.tags.append(tag)

    def describe(self):
        return f"{self.title} | {self.category} | {self.difficulty}"
```

Usage:

```
resource = Resource(
    "Python OOP",
    "Python",
    "Intermediate"
)

resource.add_tag("OOP")
resource.add_tag("Python")

print(resource.describe())
print(resource.tags)
```

Output:

```
Python OOP | Python | Intermediate
['OOP', 'Python']
```

This demonstrates an important point:

> Dataclasses can contain both data and behavior.

---

41. Converting a Dataclass to a Dictionary

The `dataclasses` module provides `asdict()`.

```
from dataclasses import dataclass, asdict


@dataclass
class Resource:
    title: str
    category: str


resource = Resource("Python Basics", "Python")

data = asdict(resource)

print(data)
```

Output:

```
{'title': 'Python Basics', 'category': 'Python'}
```

This is useful when preparing data for:

- JSON

- APIs
- databases
- serialization

---

42. astuple()

Dataclasses can also be converted to tuples.

```
from dataclasses import dataclass, astuple


@dataclass
class Point:
    x: int
    y: int


point = Point(10, 20)

print(astuple(point))
```

Output:

```
(10, 20)
```

---

43. fields()

You can inspect the fields of a dataclass.

```
from dataclasses import dataclass, fields


@dataclass
class Resource:
    title: str
    category: str
    difficulty: str


for field_info in fields(Resource):
    print(field_info.name)
```

Output:

```
title
category
difficulty
```

This is useful for generic tools and serialization systems.

---

44. replace()

The `replace()` function creates a new dataclass object with selected fields changed.

```
from dataclasses import dataclass, replace


@dataclass
class Resource:
    title: str
    difficulty: str
```

```
resource1 = Resource("Python Basics", "Beginner")

resource2 = replace(
    resource1,
    difficulty="Intermediate"
)
```

Now:

```
print(resource1)
print(resource2)
```

The original remains unchanged.

This is especially useful when working with immutable or frozen-style data models.

---

45. Dataclasses and JSON

Dataclasses do not automatically turn themselves into JSON strings.

But you can convert them to dictionaries first:

```
from dataclasses import dataclass, asdict
import json


@dataclass
class Resource:
    title: str
```

```
    category: str


resource = Resource("Python Basics", "Python")

json_data = json.dumps(asdict(resource))

print(json_data)
```

Output:

```
{"title": "Python Basics", "category": "Python"}
```

This creates a useful bridge between Python objects and JSON data.

---

46. Dataclasses and Type Checking

Because dataclasses use type annotations, they work well with tools such as static type checkers.

Example:

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

A type checker can identify suspicious code such as:

```
user = User("Hafsa", "twenty five")
```

Remember:

```
```

> Type annotations themselves do not enforce runtime types.

For runtime validation, use explicit checks or an appropriate validation library when needed.

---

47. Common Mistake: Mutable Defaults

Avoid:

```
@dataclass
class Resource:
    tags: list[str] = []
```

Use:

```
@dataclass
class Resource:
    tags: list[str] = field(default_factory=list)
```

This ensures each object gets its own list.

---

48. Common Mistake: Thinking Type Annotations Validate Data

This:

```
@dataclass
class User:
    age: int
```

does not automatically prevent:

```
User("Hafsa", "hello")
```

If runtime validation is required, implement it:

```
def __post_init__(self):
    if not isinstance(self.age, int):
        raise TypeError("age must be an integer")
```

---

49. Common Mistake: Using `frozen=True` for Security

This:

```
@dataclass(frozen=True)
class User:
    password: str
```

does not encrypt or secure the password.

`frozen=True` only restricts normal attribute assignment.

Security and immutability are different concepts.

---

50. Common Mistake: Using Dataclasses for Everything

Not every class should be a dataclass.

For example, a complex service:

```
class PaymentService:
    ...
```

may have substantial behavior and very little stored state.

A dataclass may not add much value.

Use dataclasses where their data-modeling benefits make the design clearer.

---

51. Problem Solving Framework

Before converting a class into a dataclass, ask:

Step 1: Is the class mainly storing data?

If yes, continue.

Step 2: Are its fields clearly defined?

For example:

```
title
category
url
```

Step 3: Would generated `__init__`, `__repr__`, and `__eq__` be useful?

If yes, a dataclass may be appropriate.

Step 4: Are there mutable defaults?

Use:

`field(default_factory=...)`

Step 5: Does the class require validation?

Use:

`__post_init__()`

Step 6: Does the class need immutability?

Consider:

`@dataclass(frozen=True)`

Step 7: Does it need strict memory/layout behavior?

Consider:

`@dataclass(slots=True)`

---

52. Mini Project: Resource Catalog

Build a small resource catalog using dataclasses.

Requirements:

Create:

`@dataclass`

`class Resource:`

Fields:

- title
- category
- difficulty
- URL
- tags

The class should:

1. use `default_factory` for tags
2. have an `add_tag()` method
3. have a `describe()` method
4. convert itself to a dictionary
5. validate that title is not empty

Example:

```
from dataclasses import dataclass, field, asdict


@dataclass
class Resource:
    title: str
    category: str
    difficulty: str
    url: str
    tags: list[str] = field(default_factory=list)

    def __post_init__(self):
        if not self.title.strip():
```

```
            raise ValueError("Title cannot be empty")

    def add_tag(self, tag):
        if tag not in self.tags:
            self.tags.append(tag)

    def describe(self):
        return f"{self.title} | {self.category} | {self.difficulty}"

    def to_dict(self):
        return asdict(self)
```

Usage:

```
resource = Resource(
    title="Python Dataclasses",
    category="Python",
    difficulty="Intermediate",
    url="/resources/python-dataclasses"
)

resource.add_tag("python")
resource.add_tag("oop")

print(resource.describe())
print(resource.to_dict())
```

---

53. Five Minute Challenge

Create a dataclass called:

```
Book
```

with:

`title`
`author`
`price`

Add:

`discount_price()`

The method should return the price after a given percentage discount.

Then create three books and print their details.

Bonus:

- add `genre`
- use a default value
- reject negative prices using `__post_init__()`

---

54. Exercises

Exercise 1: Student

Create:

```
@dataclass
class Student:
```

Fields:

- name
- age
- grade

Add:

`is_passing()`

Exercise 2: Product

Create:

`Product`

Fields:

- name
- price
- category
- tags

Use:

`field(default_factory=list)`

Exercise 3: Coordinate

Create:

`Point`

with:

`x`
`y`

Add:

`distance_from_origin()`

Exercise 4: User

Create:

`User`

with:

`username`
`email`
`age`

Use `__post_init__()` to:

- remove whitespace
- convert email to lowercase
- reject negative age

Exercise 5: Resource

Create a haas.dev-style:

`Resource`

with:

`title`
`category`
`difficulty`
`tags`

Add:

`add_tag()`
`describe()`

Then create several resources and store them in a list.

---

55. Mini Quiz

Question 1

Which module provides dataclasses?

A. `classes`
B. `dataclasses`
C. `objects`
D. `models`

Answer: B

Question 2

Which decorator creates a dataclass?

A. `@class`
B. `@data`
C. `@dataclass`
D. `@record`

Answer: C

Question 3

Which method is automatically generated for initialization?

A. `__start__()`
B. `__create__()`
C. `__init__()`
D. `__build__()`

Answer: C

Question 4

What should you use for a mutable list default?

A. `[]`
B. `field(default_factory=list)`
C. `list()` outside the class
D. `mutable=True`

Answer: B

Question 5

What does `__post_init__()` do?

A. Runs before class definition
B. Runs after the generated `__init__()`
C. Deletes the object
D. Converts the object to JSON

Answer: B

Question 6

Does `frozen=True` make nested mutable objects deeply immutable?

A. Yes
B. No

Answer: B

---

56. Interview Questions

Beginner

1. What is a dataclass?

A Python class designed primarily for storing structured data, with common methods generated automatically.

2. Which module provides dataclasses?

The `dataclasses` module.

3. What does `@dataclass` do?

It transforms a class into a dataclass and can automatically generate methods such as `__init__`, `__repr__`, and `__eq__`.

4. Are dataclasses still normal Python classes?

Yes.

Intermediate

5. What is `field(default_factory=list)` used for?

It creates a separate list for each instance and avoids shared mutable default values.

6. What is `__post_init__()`?

A method called automatically after the generated `__init__()` method, commonly used for validation and additional initialization.

7. What does `frozen=True` do?

It prevents normal reassignment of dataclass fields after object creation.

8. What does `asdict()` do?

It converts a dataclass instance into a dictionary representation.

9. Can dataclasses contain methods?

Yes.

Advanced

10. Are dataclasses a replacement for normal classes?

No. They are particularly useful for data-focused classes but do not eliminate the need for normal classes.

11. Are type annotations in dataclasses runtime validation?

No. Type annotations describe expected types but do not automatically enforce them.

12. What is `slots=True`?

It creates a slotted dataclass, restricting arbitrary instance attributes and potentially reducing per-instance memory usage.

13. What is the difference between `frozen=True` and true deep immutability?

`frozen=True` prevents normal reassignment of the dataclass's fields, but mutable objects stored inside those fields can still be changed.

14. When would you choose a dataclass instead of a dictionary?

When the data has a known, stable structure and benefits from explicit fields, type annotations, generated methods, and reusable behavior.

---

57. Cheat Sheet

Basic Dataclass

```
from dataclasses import dataclass


@dataclass
class Resource:
    title: str
    category: str
```

Default Value

```
@dataclass
class Resource:
    title: str
    category: str = "General"
```

Mutable Default

```
from dataclasses import field


tags: list[str] = field(default_factory=list)
```

Validation

```
def __post_init__(self):
    if not self.title:
        raise ValueError("Title required")
```

Frozen

```
@dataclass(frozen=True)
class Point:
    x: int
```

```
    y: int
```

Slots

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

Keyword Only

```
@dataclass(kw_only=True)
class Resource:
    title: str
    category: str
```

Convert to Dictionary

```
from dataclasses import asdict

data = asdict(resource)
```

Convert to Tuple

```
from dataclasses import astuple

data = astuple(resource)
```

Replace Fields

```
from dataclasses import replace

new_resource = replace(
    resource,
    category="Python"
)
```

Inspect Fields

```
from dataclasses import fields

fields(Resource)
```

---

58. Final Mental Model

Think about a dataclass as:

```
          DATACLASS
             |
    ┌────────┼────────┐
    ▼        ▼        ▼
  Fields   Methods  Generated
                    behavior
    |               |
    ▼               ▼
  Data          __init__
                __repr__
                __eq__
```

The most important idea is:

> **Use a dataclass when a class is primarily a structured piece of data and you want Python to handle the repetitive parts of creating and comparing those objects.**

Dataclasses do not remove the concepts of classes, objects, methods, inheritance, or encapsulation.

They provide a cleaner way to build **data-focused classes**.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app