Dataclasses and Typed Structures

Structured Data with Type Safety

Effective Programming Practices for Economists

The Problem with Dicts

```python
# Untyped dictionary - easy to make mistakes
person = {
    "name": "Alice",
    "age": 30,
    "emial": "alice@example.com",  # Typo! No error.
}

def greet(person: dict) -> str:
    # What keys exist? What types are the values?
    return f"Hello, {person['name']}"
```

Issues:

- Typos in keys are silent errors
- No IDE autocomplete
- No documentation of structure

Dataclasses to the Rescue

```python
from dataclasses import dataclass

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

# Now this is a type error!
alice = Person(name="Alice", age=30, emial="...")  # IDE catches typo

def greet(person: Person) -> str:
    return f"Hello, {person.name}"  # Autocomplete works!
```

Dataclass Features

```python
from dataclasses import dataclass

@dataclass
class Point:
    x: float
    y: float

# Automatically generated:
p1 = Point(1.0, 2.0)           # __init__
p2 = Point(x=1.0, y=2.0)       # Keyword args work
print(p1)                       # __repr__: Point(x=1.0, y=2.0)
p1 == Point(1.0, 2.0)          # __eq__: True
```

Default Values

```python
from dataclasses import dataclass

@dataclass
class Config:
    host: str = "localhost"
    port: int = 8080
    debug: bool = False

# Use defaults
config1 = Config()                      # All defaults
config2 = Config(host="prod.example.com", debug=True)
```

Rule: Fields with defaults must come after fields without defaults

Mutable Default Gotcha

```python
from dataclasses import dataclass, field

@dataclass
class BadExample:
    items: list[int] = []  # ERROR! Mutable default

@dataclass
class GoodExample:
    items: list[int] = field(default_factory=list)  # Correct!

# Each instance gets its own list
a = GoodExample()
b = GoodExample()
a.items.append(1)
print(b.items)  # [] - not affected
```

Frozen Dataclasses (Immutable)

```python
from dataclasses import dataclass

@dataclass(frozen=True)
class Coordinates:
    latitude: float
    longitude: float

# Can be used as dict key!
location = Coordinates(52.52, 13.405)
locations: dict[Coordinates, str] = {location: "Berlin"}

# Cannot be modified
location.latitude = 0  # TypeError: cannot assign to field
```

Prefer `frozen=True` for data that shouldn't change

NamedTuple Alternative

```python
from typing import NamedTuple

class Point(NamedTuple):
    x: float
    y: float

# Immutable by default (it's a tuple)
p = Point(1.0, 2.0)
p.x = 3.0  # Error!

# Can unpack like a tuple
x, y = p

# Can index like a tuple
first = p[0]
```

Dataclass vs NamedTuple

Feature	Dataclass	NamedTuple
Mutable	Yes (by default)	No
Inheritance	Full support	Limited
Memory	Standard	Slightly less
Hashable	Only if frozen	Always
Tuple unpacking	No	Yes
Dict-like methods	No	Yes

Rule of thumb:

TypedDict for Dict Shapes

```python
from typing import TypedDict

class PersonDict(TypedDict):
    name: str
    age: int
    email: str

# Type checker validates keys and value types
person: PersonDict = {
    "name": "Alice",
    "age": 30,
    "email": "alice@example.com",
}

# Error: missing key
bad: PersonDict = {"name": "Bob"}  # Type error!
```

Optional Keys in TypedDict

```python
from typing import TypedDict, NotRequired

class Config(TypedDict):
    host: str
    port: int
    debug: NotRequired[bool]  # Optional key

# Both are valid:
config1: Config = {"host": "localhost", "port": 8080}
config2: Config = {"host": "localhost", "port": 8080, "debug": True}
```

When to Use Each

Use `@dataclass` when:

- Creating new data structures
- Need methods on the data
- Want IDE autocomplete for attributes
- Might need to modify values

Use `TypedDict` when:

- Working with JSON data
- Interfacing with APIs that return dicts
- Gradual typing of existing dict-based code

Complex Nested Structures

```python
from dataclasses import dataclass

@dataclass(frozen=True)
class Address:
    street: str
    city: str
    country: str

@dataclass(frozen=True)
class Person:
    name: str
    age: int
    address: Address  # Nested dataclass

# Create nested structures
alice = Person(
    name="Alice",
    age=30,
    address=Address(
        street="123 Main St",
        city="Berlin",
        country="Germany",
```

Dataclass with Methods

```python
from dataclasses import dataclass

@dataclass
class Rectangle:
    width: float
    height: float

    def area(self) -> float:
        return self.width * self.height

    def scale(self, factor: float) -> "Rectangle":
        return Rectangle(
            width=self.width * factor,
            height=self.height * factor,
        )
```

Post-Init Processing

```python
from dataclasses import dataclass, field

@dataclass
class Circle:
    radius: float
    area: float = field(init=False)  # Computed, not in __init__

    def __post_init__(self) -> None:
        """Called after __init__."""
        import math
        self.area = math.pi * self.radius ** 2

circle = Circle(radius=5)
print(circle.area)  # 78.54...
```

Validation in Dataclasses

```python
from dataclasses import dataclass

@dataclass
class PositiveNumber:
    value: float

    def __post_init__(self) -> None:
        if self.value <= 0:
            raise ValueError(f"Value must be positive, got {self.value}")

PositiveNumber(5)   # OK
PositiveNumber(-1)  # Raises ValueError
```

For complex validation, consider Pydantic or attrs

Real Example: Model Configuration

```python
from dataclasses import dataclass, field

@dataclass(frozen=True)
class ModelConfig:
    """Configuration for an economic model."""
    n_periods: int
    discount_factor: float = 0.95
    interest_rate: float = 0.03
    grid_points: int = 100
    seed: int = 12345

# Type-safe configuration
config = ModelConfig(n_periods=50, grid_points=200)
```

Summary

Structured data options:

- `@dataclass` - Flexible, feature-rich, mutable by default
- `NamedTuple` - Immutable, tuple-like, lightweight
- `TypedDict` - Type-safe dictionaries

Best practices:

- Use `frozen=True` for immutable data
- Use `field(default_factory=...)` for mutable defaults
- Nest dataclasses for complex structures