Advanced Type Patterns

NewType, Literal, Protocol, and More

Effective Programming Practices for Economists

The Problem: Semantically Different Types

```python
def assign_driver(car_id: int, driver_id: int) -> None:
    ...

# Oops! Swapped arguments - no type error!
assign_driver(driver_id, car_id)
```

Both are `int`, but they mean different things

How do we prevent accidental swapping?

NewType: Semantic Distinction

```python
from typing import NewType

CarId = NewType("CarId", int)
DriverId = NewType("DriverId", int)

def assign_driver(car_id: CarId, driver_id: DriverId) -> None:
    ...

# Now this is a type error!
car: CarId = CarId(123)
driver: DriverId = DriverId(456)
assign_driver(driver, car)  # Error: wrong order!
```

NewType Details

```python
from typing import NewType

UserId = NewType("UserId", int)

# Creating values
user_id = UserId(3)  # This is just 3 at runtime

# NewType is zero-cost at runtime
type(UserId(3))  # <class 'int'>

# But type checkers distinguish them
def get_user(user_id: UserId) -> str: ...

get_user(3)          # Type error!
get_user(UserId(3))  # OK
```

Literal Types

```python
from typing import Literal

def set_mode(mode: Literal["read", "write", "append"]) -> None:
    ...

set_mode("read")    # OK
set_mode("write")   # OK
set_mode("reed")    # Type error! Typo caught.
```

Use Literal when:

- You have a fixed set of valid string values
- You want typo detection
- You don’t need a full Enum

Literal vs Enum

```python
from typing import Literal
from enum import Enum

# Literal approach
Mode = Literal["read", "write", "append"]

# Enum approach
class Mode(Enum):
    READ = "read"
    WRITE = "write"
    APPEND = "append"
```

Literal: Simpler, inline, good for APIs accepting strings
Enum: More features, namespace, better for internal use

Protocol: Structural Typing

```python
from typing import Protocol

class Drawable(Protocol):
    def draw(self) -> None:
        ...

def render(item: Drawable) -> None:
    item.draw()

# Any class with a draw() method works - no inheritance needed!
class Circle:
    def draw(self) -> None:
        print("Drawing circle")

render(Circle())  # OK! Circle has draw() method
```

Protocol: Duck Typing Made Safe

```python
from typing import Protocol

class Sized(Protocol):
    def __len__(self) -> int: ...

def print_length(obj: Sized) -> None:
    print(f"Length: {len(obj)}")

# All of these work:
print_length([1, 2, 3])    # list has __len__
print_length("hello")       # str has __len__
print_length({1, 2})        # set has __len__
```

"If it walks like a duck..."

Protocols capture the interface, not the inheritance.

Protocol for Callables

```python
from typing import Protocol

class Optimizer(Protocol):
    def minimize(
        self,
        fun: Callable[[float], float],
        x0: float,
    ) -> float:
        ...

# Any object with this method signature works
def run_optimization(optimizer: Optimizer) -> float:
    return optimizer.minimize(lambda x: x**2, 1.0)
```

@overload: Type-Dependent Returns

```python
from typing import overload

@overload
def process(value: int) -> str: ...
@overload
def process(value: str) -> int: ...

def process(value: int | str) -> str | int:
    if isinstance(value, int):
        return str(value)
    return len(value)

# Type checker knows:
reveal_type(process(3))      # str
reveal_type(process("hello")) # int
```

@overload Use Cases

```python
from typing import overload

@overload
def fetch(url: str, parse: Literal[True]) -> dict: ...
@overload
def fetch(url: str, parse: Literal[False]) -> str: ...

def fetch(url: str, parse: bool = True) -> dict | str:
    response = requests.get(url).text
    if parse:
        return json.loads(response)
    return response

# Return type depends on parse argument
data: dict = fetch("...", parse=True)
raw: str = fetch("...", parse=False)
```

TypeGuard: Custom Type Narrowing

```python
from typing import TypeGuard

def is_string_list(val: list[object]) -> TypeGuard[list[str]]:
    """Check if all elements are strings."""
    return all(isinstance(x, str) for x in val)

def process(items: list[object]) -> None:
    if is_string_list(items):
        # Type checker knows items is list[str] here
        for item in items:
            print(item.upper())  # .upper() is safe!
```

Final: Prevent Overriding

```python
from typing import Final, final

# Final variable - cannot be reassigned
MAX_SIZE: Final = 100

# Final method - cannot be overridden
class Base:
    @final
    def critical_method(self) -> None:
        ...

# Final class - cannot be subclassed
@final
class Singleton:
    ...
```

ClassVar: Class-Level Attributes

```python
from typing import ClassVar
from dataclasses import dataclass

@dataclass
class Counter:
    # Shared across all instances
    count: ClassVar[int] = 0

    # Instance attribute
    name: str

    def __post_init__(self) -> None:
        Counter.count += 1
```

Annotated: Metadata on Types

```python
from typing import Annotated
from annotated_types import Gt, Le

# Add constraints as metadata
PositiveInt = Annotated[int, Gt(0)]
Percentage = Annotated[float, Gt(0), Le(100)]

def set_volume(level: Percentage) -> None:
    ...

# Used by validation libraries (Pydantic, etc.)
# Type checkers may not enforce, but runtime validators can
```

Real Example: pylcm Type Aliases

```
from jaxtyping import Float, Int
from jax import Array

# Semantic type aliases for domain concepts
type ContinuousState = Float[Array, "..."]
type DiscreteState = Int[Array, "..."]
type RegimeName = str

# Function signatures become documentation
def get_next_state(
    current: ContinuousState,
    action: DiscreteState,
) -> ContinuousState:
    ...
```

Combining Patterns

```python
from typing import NewType, Literal, Protocol
from dataclasses import dataclass

# Domain-specific IDs
ModelId = NewType("ModelId", str)

# Constrained values
Status = Literal["pending", "running", "completed", "failed"]

# Interface for runners
class ModelRunner(Protocol):
    def run(self, model_id: ModelId) -> Status: ...

@dataclass(frozen=True)
class ModelResult:
    model_id: ModelId
    status: Status
    output: dict[str, float]
```

Pattern: Make Illegal States Unrepresentable

```python
# Bad: Can create invalid combinations
@dataclass
class File:
    path: str
    is_open: bool
    content: str | None  # Only valid if is_open

# Good: Separate types for separate states
@dataclass
class ClosedFile:
    path: str

@dataclass
class OpenFile:
    path: str
    content: str

File = ClosedFile | OpenFile
```

Summary

Semantic distinction:

- `NewType` - Same runtime type, different static type
- `Literal` - Fixed set of valid values

Structural typing:

- `Protocol` - Interface without inheritance

Advanced control:

- `@overload` - Type-dependent signatures
- `TypeGuard` - Custom type narrowing
- `Final` - Prevent modification