IMEN574/IDSC574 Programing for Data Science

7. **Python**

Spring 2026

Instructor: **Sangwoo Mo**

What We'll Cover Today

This lecture assumes you know basic Python (variables, functions, loops, lists, dicts).
We'll focus on patterns that make your DS code professional, maintainable, and scalable.

OOP Foundations	Classes, inheritance, and why they matter for DS
Abstract Base Classes	Design contracts — the regression model example
Magic Methods	Making your objects behave like built-in types
Decorators	Adding functionality on top of classes and functions
Generators & Context Managers	Memory efficiency and resource safety
Modern Patterns	Type hints, dataclasses, functional programming

1

OOP Foundations

Why Object-Oriented Programming for Data Science?



Why OOP for Data Science?

Without OOP

Scattered functions with no clear structure

Global state shared between functions

Hard to test, reuse, or extend code

Copy-paste patterns everywhere

With OOP

Data + behavior bundled together

Clear interfaces between components

Easy to test, swap, and extend

scikit-learn, PyTorch, pandas — all OOP!



Almost every major DS library is built with OOP. Understanding it helps you use and extend them.

```
# scikit-learn: OOP everywhere!
model = RandomForestClassifier(n_estimators=100)   # __init__
model.fit(X_train, y_train)                        # method
predictions = model.predict(X_test)                 # method
```

Class = Blueprint, Instance = Reality

Think of a **`class`** as a cookie cutter, and an **`instance`** as the actual cookie.

```
class DataPipeline:
    """A reusable data processing pipeline."""
    def __init__(self, name, steps=None):
        self.name = name        # instance attribute
        self.steps = steps or []
    def add_step(self, func):
        self.steps.append(func)
        return self             # enables chaining
    def run(self, data):
        for step in self.steps:
            data = step(data)
        return data
```

`class`
The blueprint — defines what attributes and methods instances will have

`__init__`
The constructor — called automatically when creating an instance

`self`
Reference to the current instance — like 'this' in Java/C++

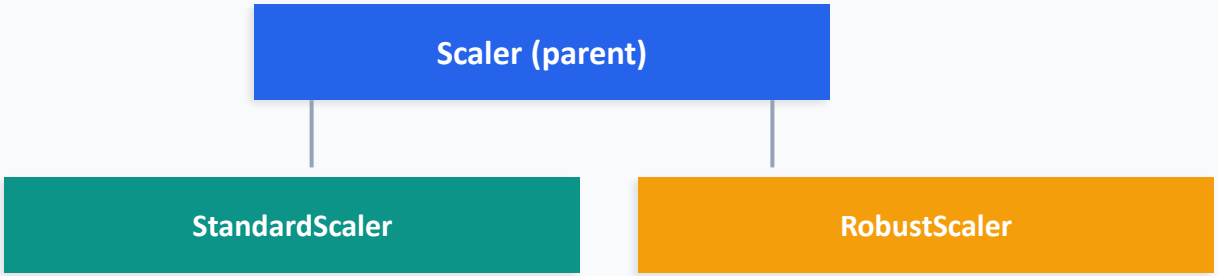
`Instance`
A concrete object created from the class blueprint



return self in methods allows method chaining: pipeline.add_step(clean).add_step(normalize).run(data)

Inheritance — Reuse & Specialize

Inheritance lets a child class reuse code from a parent class, while adding or changing behavior.




```python
class Scaler:           # Parent
    def fit(self, X):
        self.mean_ = X.mean(axis=0)
        self.std_  = X.std(axis=0)
        return self
    def transform(self, X):
        return (X - self.mean_) / self.std_
```

```python
class RobustScaler(Scaler):  # Child
    def fit(self, X):        # Override!
        self.median_ = np.median(X, 0)
        self.iqr_ = iqr(X, axis=0)
        return self
    def transform(self, X):  # Override!
        return (X - self.median_) / self.iqr_
```

super() and Method Resolution Order

```python
class BaseModel:
    def __init__(self, name):
        self.name = name
        self.is_fitted = False
class RegModel(BaseModel):
    def __init__(self, name, alpha=1.0):
        super().__init__(name)  # Call parent!
        self.alpha = alpha
model = RegModel('ridge', alpha=0.5)
print(model.name)   # 'ridge' (from parent)
print(model.alpha)  # 0.5 (from child)
```

What is super()?

super() gives you access to the parent class's methods.

Most commonly used in __init__ to make sure the parent's initialization runs too.

MRO (Method Resolution Order):
Python's rule for deciding which method to call when multiple classes define the same method. Uses C3 linearization.



Always call super().__init__() in child classes to ensure parent state is properly initialized.

```python
RegModel.__mro__  # → [RegModel, BaseModel, object]   Python checks in this order!
```

2

Abstract Base Classes

Designing Contracts Through Inheritance



The Problem: Inconsistent Interfaces

Imagine you're building a model evaluation framework. Different team members create model classes...

```
# Alice's model
class LinearModel:
    def train(self, X, y):    # 'train'
        ...
    def predict(self, X):
        ...
# Bob's model
class TreeModel:
    def fit(self, X, y):      # 'fit' !!
        ...
    def inference(self, X):   # 'inference' !!
        ...
```

What goes wrong?

Different method names for the same task

Can't write generic code that works with all models

Errors happen at runtime, deep in the pipeline

No way to enforce a standard interface

```
# This BREAKS because Bob used 'inference' instead of 'predict'
for model in [LinearModel(), TreeModel()]:
    model.fit(X, y)           # AttributeError on LinearModel!
    preds = model.predict(X)  # AttributeError on TreeModel!
```

ABC: Enforcing a Standard Interface

An Abstract Base Class defines required methods that all child classes must implement.

```python
from abc import ABC, abstractmethod
class BaseModel(ABC):
    """All models MUST implement these."""
    @abstractmethod
    def fit(self, X, y):
        """Learn from training data."""
        ...
    @abstractmethod
    def predict(self, X):
        """Generate predictions."""
        ...
```

How it works

1. **Inherit from ABC**
   Import from the abc module

2. **Mark methods with @abstractmethod**
   These MUST be implemented by children

3. **Try to instantiate = TypeError!**
   Fails immediately, not at runtime

```python
model = BaseModel()  # TypeError: Can't instantiate abstract class!
# This error happens IMMEDIATELY — not deep in your pipeline!
# You MUST implement fit() and predict() in a subclass first.
```

Building a BaseRegressor

Let's design an abstract base class for regression models with both abstract and concrete methods.

```
from abc import ABC, abstractmethod
import numpy as np
class BaseRegressor(ABC):
    """Abstract base class for all regression models."""
    @abstractmethod                    # Child MUST implement
    def fit(self, X: np.ndarray, y: np.ndarray) -> 'BaseRegressor':
        """Learn model parameters from training data."""
        ...
    @abstractmethod                    # Child MUST implement
    def predict(self, X: np.ndarray) -> np.ndarray:
        """Generate predictions for new data."""
        ...
    def score(self, X, y):             # Shared concrete method!
        """R-squared score — works for ALL regressors."""
        pred = self.predict(X)         # Calls child's predict()
        ss_res = np.sum((y - pred) ** 2)
        ss_tot = np.sum((y - y.mean()) ** 2)
        return 1 - ss_res / ss_tot
```

Implementing: LinearRegressor

The simplest concrete implementation — Ordinary Least Squares.

```
class LinearRegressor(BaseRegressor):
    """Ordinary Least Squares regression."""
    def fit(self, X, y):                    # Implementing abstract method
        X_b = np.c_[np.ones(len(X)), X]     # Add bias column
        self.coef_ = np.linalg.lstsq(
            X_b, y, rcond=None
        )[0]
        return self
    def predict(self, X):                    # Implementing abstract method
        X_b = np.c_[np.ones(len(X)), X]
        return X_b @ self.coef_
    # score() is inherited from BaseRegressor — no need to rewrite!
```

```
model = LinearRegressor()          # Works! All abstract methods implemented
model.fit(X_train, y_train).score(X_test, y_test)  # 0.87
```

Implementing: RidgeRegressor

Same interface, different algorithm — L2-regularized regression.

```
class RidgeRegressor(BaseRegressor):
    """Ridge regression with L2 regularization."""
    def __init__(self, alpha=1.0):
        self.alpha = alpha              # Regularization strength
    def fit(self, X, y):                # Implementing abstract method
        X_b = np.c_[np.ones(len(X)), X]
        n = X_b.shape[1]
        I = np.eye(n)
        I[0, 0] = 0                     # Don't regularize bias
        self.coef_ = np.linalg.solve(
            X_b.T @ X_b + self.alpha * I,
            X_b.T @ y
        )
        return self
    def predict(self, X):               # Implementing abstract method
        X_b = np.c_[np.ones(len(X)), X]
        return X_b @ self.coef_
```

Polymorphism: One Interface, Many Forms

The real power: write code once that works with ANY BaseRegressor subclass.

```
def evaluate_models(models, X_tr, y_tr,
                    X_te, y_te):
    """Works with ANY BaseRegressor!"""
    results = {}
    for model in models:
        model.fit(X_tr, y_tr)
        score = model.score(X_te, y_te)
        name = type(model).__name__
        results[name] = score
    return results
# Both models work identically!
models = [LinearRegressor(),
          RidgeRegressor(alpha=0.5)]
scores = evaluate_models(models, ...)
```

Why this matters

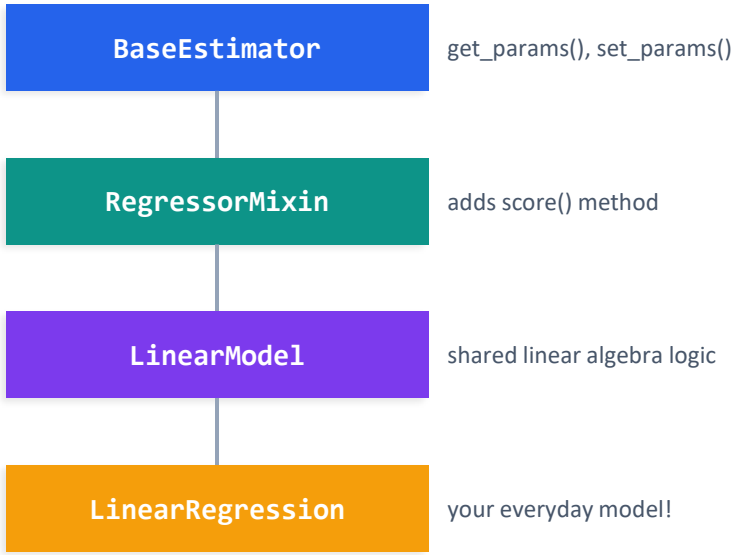
- Write evaluation code once
- Add new models without changes
- Guaranteed method availability
- Test models in isolation
- Same pattern as scikit-learn!



This is the Open/Closed Principle: open for extension (new models), closed for modification (evaluation code).

Real-World: How scikit-learn Uses This

sklearn Class Hierarchy




```python
# sklearn uses ABC + Mixin pattern
from sklearn.base import (
    BaseEstimator,    # get/set_params
    RegressorMixin,   # adds score()
    TransformerMixin, # adds fit_transform()
)
# Your custom sklearn-compatible model:
class MyModel(BaseEstimator, RegressorMixin):
    def fit(self, X, y): ...
    def predict(self, X): ...
```

What is a Mixin?

A Mixin is a small class that adds one specific feature. Unlike regular inheritance, mixins are designed to be combined.

RegressorMixin adds only score()
TransformerMixin adds only fit_transform()

3

Magic Methods

Making Your Objects Behave Like Built-in Types



What Are Magic Methods (Dunders)?

Magic methods are special methods surrounded by double underscores: **`__method__`**

Python calls them automatically when you use operators, built-in functions, or special syntax.

What you write	What Python actually calls
`len(obj)`	`obj.__len__()`
`str(obj)` or `print(obj)`	`obj.__str__()`
`repr(obj)`	`obj.__repr__()`
`obj[key]`	`obj.__getitem__(key)`
`obj(args)`	`obj.__call__(args)`
`for x in obj:`	`obj.__iter__()`
`a + b`	`a.__add__(b)`



By implementing these methods, your custom classes work seamlessly with Python's built-in functions!

__repr__ and __str__ — String Representations

The most important magic methods for debugging and usability.

```python
class RidgeModel:
    def __init__(self, alpha=1.0):
        self.alpha = alpha
        self.r2_ = None
    def __repr__(self):
        # For developers — unambiguous
        return f"RidgeModel(alpha={self.alpha})"
    def __str__(self):
        # For users — readable
        if self.r2_ is not None:
            return (f'Ridge model (alpha={self.alpha},'
                    f' R²={self.r2_:.3f})')
        return f"Ridge model (alpha={self.alpha})"
```

__repr__

Developer-oriented. Should ideally return a string that could recreate the object. Called by repr() and in the REPL.

__str__

User-oriented. Human-friendly description. Called by str() and print(). Falls back to __repr__ if not defined.

```
model = RidgeModel(alpha=0.5);  repr(model) → "RidgeModel(alpha=0.5)"
print(model) → "Ridge model (alpha=0.5)"    # uses __str__
```

Making Objects Act Like Collections

```
class Dataset:
    """A custom dataset that behaves like a list."""
    def __init__(self, features, labels):
        self.features = features
        self.labels = labels
    def __len__(self):               # len(dataset)
        return len(self.features)
    def __getitem__(self, idx):       # dataset[0], dataset[1:5]
        return self.features[idx], self.labels[idx]
    def __iter__(self):               # for x, y in dataset:
        for i in range(len(self)):
            yield self[i]             # Uses __getitem__!
    def __contains__(self, item):     # if x in dataset:
        return item in self.features
```

```
ds = Dataset(X, y);  len(ds) → 1000;  ds[0] → (features, label);  for x,y in ds: ...
```

__call__ — Making Instances Callable

When you implement __call__, your object becomes a function-like object.

```python
class Preprocessor:
    """A callable preprocessing pipeline."""
    def __init__(self, steps):
        self.steps = steps
    def __call__(self, data):
        for step in self.steps:
            data = step(data)
        return data
# Use it like a function!
preprocess = Preprocessor([clean, scale])
result = preprocess(raw_data)  # Calls __call__
```

Why is this useful?

Stateful functions

Unlike regular functions, callable objects can remember state between calls.

Compatible with functional APIs

Works anywhere a function is expected (map, filter, callbacks).

Used everywhere in ML

PyTorch nn.Module, Keras layers — all use __call__.



PyTorch: model(input) works because nn.Module implements __call__, which runs forward().

4

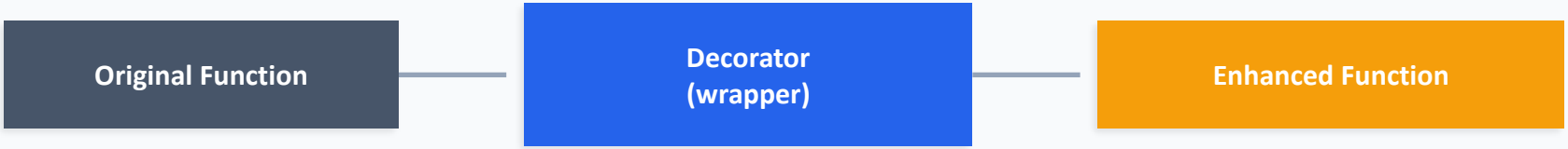
Decorators

Adding Functionality Without Modifying Code



What Is a Decorator?

A decorator **wraps a function with extra behavior** — like adding gift wrapping to a present.




```python
# The LONG way (without @):
def my_function():
    ...
my_function = decorator(my_function)
# decorator takes the function
# and returns a new, enhanced version
```

```python
# The SHORT way (with @):
@decorator
def my_function():
    ...
# Exactly the same thing!
# @ is just syntactic sugar.
```

@decorator is just syntactic sugar for: function = decorator(function)

Your First Decorator: Timer

```python
import time
import functools
def timer(func):
    """Measure how long a function takes."""
    @functools.wraps(func)  # Preserve name
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        elapsed = time.perf_counter() - start
        print(f'{func.__name__}: {elapsed:.4f}s')
        return result
    return wrapper
@timer
def train_model(X, y, epochs=100):
    """Train a model."""
    ...  # training logic
```

Step by step:

1. timer(func) receives the original function
2. It defines wrapper() inside
3. wrapper() runs code BEFORE calling func
4. wrapper() calls the original func(*args, **kwargs)
5. wrapper() runs code AFTER getting the result
6. timer() returns wrapper as the replacement

```python
train_model(X, y)  # Output: "train_model: 2.3456s"   (then returns result)
```

Why functools.wraps Matters

```
# WITHOUT @functools.wraps
def timer(func):
    def wrapper(*args, **kw):
        ...
    return wrapper
@timer
def train(): ...
train.__name__   # "wrapper" ← WRONG!
train.__doc__    # None      ← LOST!
```

```
# WITH @functools.wraps
def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        ...
    return wrapper
@timer
def train(): ...
train.__name__   # "train" ← CORRECT!
```



Always use @functools.wraps(func) in your decorators. It preserves the original function's metadata.

```
# Real-world impact: without wraps, debugging becomes painful
import logging
logging.info(f"Calling {func.__name__}")  # "wrapper" vs "train_model"
help(train)  # Shows wrapper docs vs original docs
```

Decorator with Arguments (Factory Pattern)

What if you want to pass arguments to a decorator? You need an extra layer of nesting.

```python
def retry(max_attempts=3, delay=1.0):
    """Decorator FACTORY — returns a decorator."""
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(max_attempts):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == max_attempts - 1:
                        raise
                    time.sleep(delay)
        return wrapper
    return decorator  # Returns the real decorator
@retry(max_attempts=5, delay=2.0)
def fetch_data(url):
    return requests.get(url).json()
```

3 Levels of Nesting

`retry()`

Factory: takes the arguments (max_attempts, delay)

`decorator()`

The actual decorator: takes the function

`wrapper()`

The replacement: runs the retry logic

Practical Decorators for Data Science

```
# Built-in caching decorator
from functools import lru_cache
@lru_cache(maxsize=128)
def expensive_distance_matrix(n, seed):
    """Cache results of expensive computations.
    Only recomputes when args change!"""
    np.random.seed(seed)
    X = np.random.randn(n, 10)
    return pairwise_distances(X)
```

```
# Input validation decorator
def validate_2d(func):
    @functools.wraps(func)
    def wrapper(self, X, *args, **kw):
        if X.ndim != 2:
            raise ValueError(
                f"Expected 2D, got {X.ndim}D")
        return func(self, X, *args, **kw)
    return wrapper
```

```
# Using validation decorator on a model class
class MyModel(BaseRegressor):
    @validate_2d
    def fit(self, X, y):
        self.n_features_ = X.shape[1]
        ...  # training logic
    @validate_2d
    def predict(self, X):
        ...  # prediction logic
```

@property — Controlled Attribute Access

@property turns a method into a read-only attribute — adding validation or computation transparently.

```python
class ModelConfig:
    def __init__(self, lr, epochs):
        self._lr = lr          # Private by convention
        self._epochs = epochs
    @property
    def learning_rate(self):
        """Read-only access to learning rate."""
        return self._lr
    @property
    def epochs(self):
        return self._epochs
    @epochs.setter
    def epochs(self, value):
        if value <= 0:
            raise ValueError("Must be positive")
        self._epochs = value
```

@property (getter)

Makes a method look like an attribute. Users write cfg.learning_rate instead of cfg.get_learning_rate().

@name.setter

Optional: allows assignment with validation. cfg.epochs = 50 triggers the setter, which can validate.

```
cfg = ModelConfig(0.001, 100);  cfg.learning_rate → 0.001;  cfg.epochs = -5 → ValueError!
```

@classmethod and @staticmethod

```python
class Experiment:
    _count = 0  # Class variable
    def __init__(self, name, params):
        self.name = name
        self.params = params
        Experiment._count += 1
    @classmethod
    def from_config(cls, config_dict):
        """Alternative constructor."""
        return cls(
            config_dict['name'],
            config_dict['params']
        )
    @staticmethod
    def validate_params(params):
        """Utility — no self or cls needed."""
        return all(v > 0 for v in params.values())
```

@classmethod

Gets **cls** (the class) as first argument instead of self.

Most common use: **alternative constructors** (from_config, from_json, from_csv).

@staticmethod

No self or cls — just a regular function that lives inside the class namespace.

Use for **utility functions** that logically belong to the class but don't need instance/class state.

```python
exp = Experiment.from_config({'name': 'v1', 'params': {'lr': 0.01}})  # classmethod!
```

Decorating Entire Classes

Decorators can also be applied to classes — adding behavior to all methods at once.

```
import logging
def add_logging(cls):
    """Class decorator: adds logging to all public methods."""
    logger = logging.getLogger(cls.__name__)
    for name, method in list(vars(cls).items()):
        if callable(method) and not name.startswith('_'):
            def make_logged(m, n):          # Closure captures m, n
                @functools.wraps(m)
                def logged(self, *args, **kwargs):
                    logger.info(f"Calling {n}")
                    return m(self, *args, **kwargs)
                return logged
            setattr(cls, name, make_logged(method, name))
    return cls
@add_logging
class MyPipeline:
    def fit(self, X, y):    ...   # Automatically logged!
    def predict(self, X):   ...   # Automatically logged!
```

Decorator Summary

Decorator	What it does	Use case
`@timer`	Measures execution time	Performance profiling
`@lru_cache`	Caches function results	Expensive computations
`@retry`	Retries on failure	API calls, I/O
`@validate_input`	Checks function arguments	Data validation
`@property`	Method → attribute access	Getters/setters
`@classmethod`	Class as first argument	Alt constructors
`@staticmethod`	No self or cls	Utility functions
`@add_logging`	Wraps all class methods	Debugging, monitoring



Think of decorators as Lego blocks: stack them to build powerful, reusable functionality.

5

Generators & Context Managers

Memory Efficiency and Resource Safety



What Is a Generator?

A generator is a function that **produces values one at a time** using `yield` instead of return.

```
# Regular function: ALL at once
def get_squares(n):
    result = []
    for i in range(n):
        result.append(i ** 2)
    return result  # Entire list in memory!
squares = get_squares(1_000_000)
# 1M items stored at once!
```

```
# Generator: ONE at a time
def gen_squares(n):
    for i in range(n):
        yield i ** 2    # Pause & produce
squares = gen_squares(1_000_000)
# Nothing computed yet! (lazy)
```

Memory: O(n) — stores entire list

Memory: O(1) — one value at a time!



yield pauses the function and saves its state. Next call resumes from where it left off.

```
for sq in gen_squares(1_000_000):  print(sq)  # Computes each value only when needed
```

Generators in Data Science: Mini-Batch Training

```python
def batch_generator(X, y, batch_size=32):
    """Yield mini-batches for training."""
    n = len(X)
    indices = np.arange(n)
    np.random.shuffle(indices)
    for start in range(0, n, batch_size):
        idx = indices[start:start+batch_size]
        yield X[idx], y[idx]  # One batch
# Training loop
for epoch in range(100):
    for X_batch, y_batch in batch_generator(
            X_train, y_train, batch_size=64):
        model.partial_fit(X_batch, y_batch)
```

Why generators here?

- Only one batch in memory at a time
- Perfect for datasets > RAM
- Easy to add augmentation
- Compose with other generators
- Same pattern as PyTorch DataLoader

```python
# Generator expressions — even more concise
total = sum(x**2 for x in data if x > 0)   # No list created!
avg = np.mean([score for m in models for score in m.cv_scores_])
```

Context Managers — The with Statement

Context managers guarantee cleanup code runs, even if an error occurs.

```
# WITHOUT context manager:
f = open('data.csv')
try:
    data = f.read()
    process(data)
finally:
    f.close()  # Must remember this!
```

```
# WITH context manager:
with open('data.csv') as f:
    data = f.read()
    process(data)
# File automatically closed!
# Even if an exception occurs!
```

`with` calls `__enter__()` at the start and `__exit__()` at the end — guaranteed, even on exceptions.

```
# You already use context managers every day!
with open('data.csv') as f: ...          # File I/O
with db.connect() as conn: ...           # Database connections
with torch.no_grad(): ...                # PyTorch inference mode
```

Creating Your Own Context Managers

```
# Method 1: Class-based
class Timer:
    def __enter__(self):
        self.start = time.perf_counter()
        return self
    def __exit__(self, *exc_info):
        self.elapsed = (
            time.perf_counter() - self.start)
        print(f'Elapsed: {self.elapsed:.3f}s')
        return False  # Don't suppress exceptions
```

```
# Method 2: Generator-based (easier!)
from contextlib import contextmanager
@contextmanager
def timer():
    start = time.perf_counter()
    yield  # <- Code inside 'with' runs here
    elapsed = time.perf_counter() - start
    print(f'Elapsed: {elapsed:.3f}s')
```

```
# Usage (both work the same!)
with Timer() as t:
    model.fit(X, y)
# Output: Elapsed: 2.345s
```

```
# Usage (generator version)
with timer():
    model.fit(X, y)
# Output: Elapsed: 2.345s
```



Use @contextmanager for simple cases — it's much less boilerplate than the class-based approach.

Context Managers in Practice

```
# Temporary directory for experiment artifacts
import tempfile
from pathlib import Path
with tempfile.TemporaryDirectory() as tmpdir:
    model_path = Path(tmpdir) / "model.pkl"
    joblib.dump(model, model_path)
    metrics = evaluate(model_path)
    log_metrics(metrics)
# Directory and all files automatically deleted!
```

```
# Custom: Seed context for reproducible experiments
@contextmanager
def reproducible(seed=42):
    state = np.random.get_state()
    np.random.seed(seed)
    try:
        yield
    finally:
        np.random.set_state(state)  # Restore original state
```

6

Modern Python Patterns

Type Hints, Dataclasses & Functional Programming



Type Hints — Self-Documenting Code

Type hints don't change runtime behavior — they're documentation that tools can verify.

```
# WITHOUT type hints:
def preprocess(df, columns, fill):
    # What is df? A DataFrame?
    # columns? A list? A string?
    # fill? A number? A string?
    ...
```

Ambiguous — must read the code to understand

```
# WITH type hints:
def preprocess(
    df: pd.DataFrame,
    columns: list[str],
    fill: float | None = None
) -> pd.DataFrame:
    ...
```

Clear — signature tells you everything

```
# Common type hint patterns for DS
from typing import Optional, Union, Callable
scores: list[float] = [0.85, 0.91, 0.88]
params: dict[str, float] = {'lr': 0.01, 'alpha': 0.5}
callback: Optional[Callable] = None  # Same as: Callable | None
```

Advanced Type Hints & Protocol

```python
from typing import Protocol
class Fittable(Protocol):
    """Structural typing — no inheritance!"""
    def fit(self, X, y) -> 'Fittable': ...
    def predict(self, X) -> np.ndarray: ...
def evaluate(model: Fittable, X, y):
    """Accepts ANY object with fit & predict."""
    model.fit(X, y)
    return model.predict(X)
```

Protocol vs ABC

ABC
Must explicitly inherit. Runtime check. Enforced at instantiation.

Protocol
No inheritance needed! Static check only. Duck typing + type safety.



Protocol = 'if it looks like a duck and quacks like a duck' — but with type checker support!

```python
# Protocol works with ANY class that has the right methods — no inheritance required!
class SklearnModel:  # NOT inheriting from Fittable
    def fit(self, X, y): ...    # Has the right methods → Protocol satisfied!
    def predict(self, X): ...
```

Dataclasses — Less Boilerplate, More Data

@dataclass automatically generates __init__, __repr__, __eq__ and more.

```python
# WITHOUT dataclass (so much typing!):
class Config:
    def __init__(self, name, lr, epochs):
        self.name = name
        self.lr = lr
        self.epochs = epochs
    def __repr__(self):
        return (f'Config(name={self.name},'
                f' lr={self.lr},'
                f' epochs={self.epochs})')
    def __eq__(self, other): ...
```

```python
# WITH dataclass (so clean!):
from dataclasses import dataclass
@dataclass
class Config:
    name: str
    lr: float = 0.001
    epochs: int = 100
# __init__, __repr__, __eq__
# all auto-generated!
```

```python
cfg = Config(name="baseline", lr=0.01)
print(cfg)  # Config(name='baseline', lr=0.01, epochs=100)
cfg2 = Config(name='baseline', lr=0.01)
cfg == cfg2  # True!  (auto-generated __eq__)
```

Dataclass Advanced Features

```python
from dataclasses import dataclass, field
@dataclass(frozen=True)  # Immutable!
class ModelParams:
    n_features: int
    n_classes: int
    weights: tuple[float, ...]
params = ModelParams(10, 3, (0.1, 0.9))
params.n_features = 5  # FrozenInstanceError!
# Can use as dict key (hashable)!
```

```python
@dataclass
class ExperimentConfig:
    name: str
    lr: float = 0.001
    tags: list[str] = field(
        default_factory=list  # Mutable default!
    )
    notes: str | None = None
# field(default_factory=list) avoids
# the mutable default argument trap
```

Warning: Never use mutable defaults like `tags: list = []` — use `field(default_factory=list)` instead!

```python
# Useful options:  @dataclass(frozen=True)   → immutable (hashable, dict key ok)
#                  @dataclass(order=True)    → comparison operators (<, >, etc.)
#                  @dataclass(slots=True)    → faster, less memory (Python 3.10+)
```

Functional Programming Basics

Python supports functional programming patterns alongside OOP.

```
# Lambda: anonymous functions
square = lambda x: x ** 2
# map: apply function to all items
scores = list(map(lambda m: m.score(X,y),
                  models))
# filter: keep items matching condition
good = list(filter(lambda s: s > 0.8, scores))
```

```
# But... list comprehensions are more Pythonic!
scores = [m.score(X,y) for m in models]
good = [s for s in scores if s > 0.8]
# Cleaner, more readable, same performance
# Prefer comprehensions over map/filter
# in most cases!
```



List comprehensions are generally preferred over map/filter in Python — use lambda sparingly.

```
# functools.partial: fix some arguments of a function
from functools import partial
normalize = partial(np.linalg.norm, axis=1)  # Fix axis=1
norms = normalize(X)  # Same as: np.linalg.norm(X, axis=1)
```

Closures — Functions That Remember

A closure is a function that captures and remembers variables from its enclosing scope.

```
def make_scaler(mean, std):
    """Returns a scaling function that
    remembers the training statistics."""
    def scale(X):              # Inner function
        return (X - mean) / std  # Uses outer vars!
    return scale               # Return the function
# Create a scaler with training stats
scaler = make_scaler(
    X_train.mean(axis=0),
    X_train.std(axis=0)
)
X_test_scaled = scaler(X_test)  # Uses train stats!
```

How it works

1. make_scaler() creates scale() inside

2. scale() references mean and std from the outer scope

3. When make_scaler() returns, scale() keeps a reference to those variables

4. The closure 'closes over' the free variables

This is the same mechanism that makes decorators work!



Closures power decorators, callbacks, and factory functions. Think of them as lightweight stateful objects.

Composing Functions

```
from functools import reduce
def compose(*funcs):
    """Compose functions: compose(f, g, h)(x) == f(g(h(x)))"""
    return reduce(lambda f, g: lambda x: f(g(x)), funcs)
# Build a transformation pipeline from simple functions
preprocess = compose(normalize, scale, impute, clean)
# clean → impute → scale → normalize   (right-to-left!)
result = preprocess(raw_data)  # Applies all four steps
```

```
# Alternative: pipe (left-to-right, more readable)
def pipe(*funcs):
    """pipe(f, g, h)(x) == h(g(f(x)))  — left-to-right"""
    def pipeline(x):
        for func in funcs:
            x = func(x)
        return x
    return pipeline
```

Putting It All Together

A realistic example combining ABC, decorators, dataclasses, and type hints.

```
from abc import ABC, abstractmethod
from dataclasses import dataclass
from functools import wraps
from typing import Protocol
@dataclass(frozen=True)                          # Immutable config
class TrainConfig:
    lr: float = 0.01
    epochs: int = 100
    batch_size: int = 32
def validate_input(func):                         # Decorator
    @wraps(func)
    def wrapper(self, X, *args, **kwargs):
        assert X.ndim == 2, 'Input must be 2D'
        return func(self, X, *args, **kwargs)
    return wrapper
class BaseModel(ABC):                             # Abstract base
    def __init__(self, config: TrainConfig):      # Type hint
        self.config = config
    @abstractmethod
    def fit(self, X, y): ...
    @validate_input                               # Decorator on method
    def predict(self, X): ...
```

Python Best Practices for DS

Use ABCs for shared interfaces	When building model/pipeline frameworks, define contracts
Prefer composition over deep inheritance	Mixins + shallow hierarchies beat 5-level deep chains
Implement __repr__ on every class	Future-you will thank you when debugging at 2 AM
Use decorators for cross-cutting concerns	Timing, logging, validation — write once, apply everywhere
Type hints are documentation	Even without mypy, they make code more readable
Dataclasses for data containers	Stop writing boilerplate __init__ for config/parameter objects
Generators for large data	Process files/batches lazily to avoid memory issues

Key Takeaways



OOP powers every major DS library — understanding it is essential, not optional.



ABCs enforce contracts — define interfaces once, implement many, fail fast.



Magic methods make objects Pythonic — enable len(), indexing, iteration on custom classes.



Decorators add features non-invasively — timing, caching, validation without touching core logic.



Generators process big data lazily — O(1) memory for streaming, perfect for mini-batches.



Modern patterns reduce boilerplate — type hints document, dataclasses simplify, closures compose.