Python Course 2: Object-Oriented Programming in Python

Patrick Munroe

GERAD

2025-10-22

Introduction

- **Objectives**:
  - **Theory:** Learn the principles of object-oriented programming (OOP)
  - **Practice:** Apply OOP features in Python code
- **Prerequisites**: Familiarity with imperative programming in Python
- **Pedagogical approach**: We will learn OOP in Python by refactoring an imperative program that solves an optimization problem

Plan

What is object-oriented programming?

- **Object-oriented programming** is a way to organize code using "objects" that combine data and behavior.
- How does OOP compare to imperative programming?
  - **OOP:** Code is built around objects and their interactions:
    - Create objects (from classes)
    - Call methods on objects to make them perform actions
  - **Imperative programming:** Code is organized as step-by-step instructions that manipulate data directly:
    - Modify variables
    - Use loops and conditionals

Why object-oriented programming in general?

- Helps make **modular code**: easier to split into logical pieces.
- Encourages **code reuse**: objects and classes can be reused in different programs.
- Supports **encapsulation**: logic and data stay together, reducing errors.
- Enables **abstraction**: hides unnecessary details and exposes only essential features.
- Makes programs easier to **maintain** and **extend** as requirements change.
- Models **real-world problems** by representing entities as objects.

Why object-oriented programming for operations research and data science?

- Projects often start from scratch—OOP provides a clear **framework for code design**.
- Code for optimization models and algorithms often grows quickly; OOP keeps code **organized and maintainable**.
- **Comparing and benchmarking algorithms** is common; OOP makes it easy to swap and extend implementations.
- **Complex systems** (networks, supply chains) can be modeled as interacting objects.
- OOP integrates smoothly with **Python libraries** for optimization and data science.
- Well-structured code supports **collaboration, reproducibility, and sharing**.

Transportation problem

A motivating example for learning Python

Goal: Distribute goods from suppliers to customers at minimal cost.

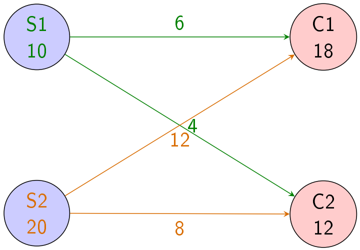
Given:

- Supplier capacities
- Customer demands
- Shipping costs

Constraints:

- Supply limits per supplier
- Demand requirements per customer

Objective: Minimize total transportation cost



Transportation problem

Data representation

Typical data for the transportation problem:

- **Suppliers:** List with supply amounts
- **Customers:** List with demand amounts
- **Cost Matrix:** Table of shipping costs per unit from each supplier to each customer

Example:

Suppliers

- S1: 10 units
- S2: 20 units

Customers

- C1: 18 units
- C2: 12 units

Cost Matrix

	C1	C2
S1	5	7
S2	4	6

Creating a virtual environment

- Use a virtual environment to keep your project's dependencies isolated.
- To **create** the environment, in a terminal, navigate to your project folder and run:

```
python -m venv venv
```

- To **activate** the environment:
  - `venv\Scripts\activate` (Windows)
  - `source venv/bin/activate` (macOS/Linux)
- When activated, `(venv)` appears at the prompt.
- To **exit** the environment, type `deactivate`.

Type hints

Concept and syntax

- **Type hints** annotate variables and function signatures with types.
- **Syntax:**
  - Annotate variables:
    `variable: type`
  - Annotate functions:
    `def func(arg: type) -> returntype:`
- **Motivation:** Improves code readability, helps with static type checking, and catches bugs early.
- **Examples:**

```python
# Annotate a function
def multiply(x: int, y: float) -> float:
    return x * y

# Annotate a variable
value: str = "hello"
```

Type hints

Example: greedy_solve

- Add type hints to the signature of our greedy_solve function:

```
def greedy_solve(
    supply_by_sup: dict[str, float],
    demand_by_cust: dict[str, float],
    cost_by_sup_cust: dict[tuple[str, str], float],
    verbose: bool = False
) -> dict[tuple[str, str], float]:
```

- Add type hints to new variables, especially when the type is not obvious:

```
flow_by_sup_cust: dict[tuple[str, str], float] = {}
```

Type hints

Example: `read_from_csv`

- Add type hints to the signature of our `read_from_csv` function:

```
def read_from_csv(
    file: str,
    index: str | list[str],
    column: str
    ) -> dict[str | tuple, float]:
```

- The operator `|` in `type1 | type2` indicates that the type can be either `type1` or `type2`.

Type hints

Aliases

- Type hint **aliases** assign a name to complex types, making code easier to read and maintain.
- **Example:**

```
CostDict = dict[tuple[str, str], float]
FlowDict = dict[tuple[str, str], float]

def greedy_solve(
    supply_by_sup: dict[str, float],
    demand_by_cust: dict[str, float],
    cost_by_sup_cust: CostDict
    ) -> FlowDict:
    flow_by_sup_cust: FlowDict = {}
    ...
```

- Multiple aliases can refer to the same type.

Classes

- A **class** is a *blueprint* for creating objects.
- **Syntax:**

```
class ClassName:
    # members
```

- It describes what an object will be like—its data and behavior—but does not itself create any objects.
- The data (variables) and behavior (methods/functions) specified in a class are called **members** (in Python, members are also referred to as **attributes**).

Classes

Example

```python
class Supplier:
    name = None
    supply = None

class Customer:
    name = None
    demand = None
```

- Two classes are defined: `Supplier` and `Customer`.
- Each class specifies the data members that belong to each instance:
  - `Supplier`: `name` and `supply`
  - `Customer`: `name` and `demand`

Instantiation

- **Instantiation** is the process of creating a new object (called an **instance**) from a class, following its blueprint.
- **Syntax:**

```
my_object = ClassName()
```

- Each object created from a class can have its own unique data, but shares the structure and capabilities defined by the class.

Instantiation

Example

```
s1 = Supplier()
s1.name = "S1"
s1.supply = 100

s2 = Supplier()
s2.name = "S2"
s2.supply = 80

print(s1.name, s1.supply)  # S1 100
print(s2.name, s2.supply)  # S2 80
```

- Two `Supplier` objects are created.
- Their attributes are set after creation.
- Each instance has its own name and supply value.

Methods

- A **method** is a function defined inside a class.
- **Syntax:**

```
class ClassName:
    def method_name(self, ...):
        # code using self.attribute
```

- The first parameter of every instance method is `self`.
- `self` refers to the specific object on which the method is called.
- Methods use `self` to access and modify the object's attributes.
- Methods describe behaviors that objects of the class can perform.

Methods

Example: Defining methods

```
class Supplier:
    name = None
    supply = None

    def print(self) -> None:
        print(f"Supplier: {self.name}: {self.supply}")

class Customer:
    name = None
    demand = None

    def print(self) -> None:
        print(f"Customer: {self.name}: {self.demand}")
```

- Both classes define a `print` method.
- The method uses the object's own attributes.
- Each object's method produces output specific to that object.

Methods

Using methods

- You call a method by using the dot notation on an object, e.g. `s1.print()`.
- The method can access and use the object's attributes.
- This makes it easy to organize related functionality with the data it operates on.

Example:

```
s1 = Supplier()
s1.name = "S1"
s1.supply = 100
s1.print()   # Output: Supplier: S1: 100

c1 = Customer()
c1.name = "C1"
c1.demand = 50
c1.print()   # Output: Customer: C1: 50
```

Constructor

- A **constructor** is a special method called `__init__` that is automatically called when you create a new object from a class.
- **Syntax:**

```
class ClassName:
    def __init__(self, ...):
        # initialization code
```

- The constructor lets you set up the initial state (attributes) of the object.
- You can provide parameters to the constructor to initialize the object with custom values.

Constructor

Example

```python
class Supplier:
    def __init__(self, name: str,
                 supply: int | float) -> None:
        self.name = name
        self.supply = float(supply)

    def print(self) -> None:
        print(f"Supplier: {self.name}: {self.supply}")

# Creating a Supplier with initial values
s1 = Supplier("S1", 100)
s1.print()  # Output: Supplier: S1, 100.0
```

- In `__init__`, `supply` is converted to `float` to ensure `self.supply` is always a floating-point number, providing consistency for calculations and output.

Encapsulation

- **Encapsulation**: Keeping data and behavior together in a class, and protecting the internal state.
- This protection relies on **access control**, which restricts whether class members can be accessed from outside:
  - **Public**: accessible from anywhere.
  - **Private**: meant for use only inside the class.
- In Python, access control is signaled by naming conventions:
  - Public: no leading underscore (`name`)
  - Private: double leading underscores (`__name`), triggers name mangling
- Encapsulation helps keep objects safe and easy to use.

Encapsulation

Example: Private and public members

```
class Supplier:
    def __init__(self, name: str,
                 supply: int | float) -> None:
        self.__name = name            # private attribute
        self.__supply = float(supply) # private attribute

        # public method
    def print(self) -> None:
        print(f"Supplier: {self.__name}: {self.__supply}")

s1 = Supplier("S1", 100)
s1.print()            # Output: "Supplier: S1: 100.0"
print(s1.__name)      # Error: AttributeError
print(s1.__supply)    # Error: AttributeError
```

- Both `__name` and `__supply` are private and cannot be accessed directly.
- The `print` method is public and can be called to display the supplier's information.

Encapsulation

Properties

- Properties allow controlled, read-only access to private data.
- A property (getter) lets users read an attribute's value, but does not let them change it.

```python
class Supplier:
    def __init__(self, name: str,
                 supply: int | float) -> None:
        self.__name = name                # private data
        self.__supply = float(supply) # private data

    @property
    def supply(self) -> float:
        return self.__supply

s1 = Supplier("S1", 100)
print(s1.supply)          # OK: read the value
s1.supply = 42            # Error: cannot set the value
```

Encapsulation

Setters

- Setters allow controlled modification of private data through properties.
- A setter method lets users change an attribute's value safely, using custom logic if needed.

```
class Supplier:
    # ...
    @property
    def remaining_supply(self):
        return self.__remaining_supply

    @remaining_supply.setter
    def remaining_supply(self, value):
        self.__remaining_supply = value

s1 = Supplier("S1", 100)
s1.remaining_supply = 42      # OK: set the value
print(s1.remaining_supply)    # OK: now value is 42
```

Encapsulation

Setters: Check values

- Setters often include validation to enforce value constraints.

```
@remaining_supply.setter
def remaining_supply(self, value: int | float):
    if value < 0:
        raise ValueError(f"remaining_supply must be >= 0.")
    self.__remaining_supply = float(value)
```

- Assigning an invalid value raises an exception.

```
s1 = Supplier("S1", 100)
s1.remaining_supply = 42      # OK: set the value to 42.0
s1.remaining_supply = -10     # Error: remaining_supply
                              # must be >= 0.
```

Abstraction

- **Abstraction** is the process of representing complex systems by focusing on their essential features and hiding unnecessary details.
- A central way abstraction is achieved in object-oriented programming is by separating an object's **interface** from its **implementation**:
  - The **interface** specifies how users interact with the object—its accessible methods and properties.
  - The **implementation** refers to how the object's functionality is realized internally.
- This separation makes code easier to use, maintain, and extend, since changes to implementation do not affect how the object is used.

Abstraction

The `Supplier` Class

- The **interface** of `Supplier` is the set of public methods and properties that users interact with:
  - `name` (property)
  - `supply` (property)
  - `remaining_supply` (property and setter)
  - `print()` (method)
- Its **implementation** consists of:
  - Private attributes: `__name`, `__supply`, `__remaining_supply`
  - Internal workings of the constructor, properties, setter, and methods
- This separation means users can interact with the class without needing to understand or rely on its internal workings, making code easier to use and modify.

Inheritance

- **Inheritance** allows a class to reuse code from another class.
- The new class is called a **subclass** (or child class), and the original is the **superclass** (or parent class). The subclass *"is a"* type of the parent class.
- The subclass automatically gets the attributes and methods of the parent class, and can also add or override functionality.
- Inheritance promotes code reuse and makes it easier to organize and extend programs.
- **Syntax:**

```python
class Parent:
    # parent class code

class Child(Parent):
    # subclass code (inherits from Parent)
```

Inheritance

Example: Entity → Supplier

```
class Entity:
    def __init__(self, name: str) -> None:
        self.__name = name

    @property
    def name(self) -> str:
        return self.__name

class Supplier(Entity):
    def __init__(self, name: str,
                 supply: int | float) -> None:
        super().__init__(name)
        self.__supply = float(supply)
        self.__remaining_supply = float(supply)
    ...
```

- `Supplier` inherits the `name` property from `Entity`.
- The argument `name` of `Supplier.__init__` is passed to `Entity.__init__` using `super()`.

Inheritance

Example: Entity → Customer

- Similarly, `Customer` inherits from `Entity`:

```
class Customer(Entity):
    def __init__(self, name: str,
                 demand: int | float) -> None:
        super().__init__(name)
        self.__demand = float(demand)
        self.__remaining_demand = float(demand)

    @property
    def demand(self) -> float:
        return self.__demand
    # ...
```

- Customer objects have access to both inherited and new attributes/methods.

```
c1 = Customer("C1", 50)
print(c1.name)     # Inherited attribute (from Entity)
print(c1.demand)   # New attribute (from Customer)
```

Inheritance

Protected members

- Private members (`__name`) are hidden from subclasses by name mangling.
- Sometimes, subclasses should access certain internal attributes.
- **Protected members** (`_name`) can be accessed inside the class and its subclasses, but are not meant to be accessed from outside.

```python
class Entity:
    def __init__(self, name: str) -> None:
        self._name = name  # protected member

class Supplier(Entity):
    def print_name(self) -> None:
        print(self._name)  # OK: accessed from subclass
```

- **Convention:** Names starting with _ should only be accessed within the class or its subclasses, not from other code.

Inheritance

Overriding in subclasses

- **Overriding**: a subclass redefines attributes (data members or methods) from its superclass.
- The subclass version is used in subclass instances.
- To override, define an attribute with the same name in the subclass:

```python
class Parent:
    def method(self):
        print("Parent method")

class Child(Parent):
    def method(self):
        print("Child method")

p = Parent()
p.method()     # Output: Parent method
c = Child()
c.method()     # Output: Child method
```

Inheritance

Example: Overriding the print method

```
class Entity:
    # ...

    def print(self) -> None:
        print(f"Entity: {self.name}")

class Supplier(Entity):
    # ...

    def print(self) -> None:
        print(f"Supplier: {self.name}: {self.supply}")

e1 = Entity("E1")
e1.print()    # Output: Entity: E1
s1 = Supplier("S1", 100)
s1.print()    # Output: Supplier: S1: 100.0
```

- The `Supplier` subclass overrides the `print` method to display more specific information.

Composition

- **Composition** models a *"has-a"* relationship: a class contains instances of other classes as attributes.
- Unlike inheritance (*"is-a"*), composition allows us to build complex types by combining simpler ones.
- This promotes code reuse and flexibility.

Composition

Example: The Cost class

- The Cost class represents the cost from a Supplier to a Customer.
- Cost *"has a"* Supplier and a Customer as part of its state.
- This is composition, not inheritance: Cost is not a kind of Supplier or Customer, but uses them as components.

```python
class Cost:
    def __init__(self, supplier: Supplier,
                 customer: Customer,
                 value: int | float) -> None:
        self.__supplier = supplier     # composed object
        self.__customer = customer     # composed object
        self.__value = float(value)
```

Composition

Example: Using the Cost class

```
supplier = Supplier("S1", 100)
customer = Customer("C1", 50)
cost = Cost(supplier, customer, 25.0)

print(cost.supplier.name)      # Output: S1
print(cost.customer.name)      # Output: C1
print(cost.value)              # Output: 25.0
```

- Cost objects store and use instances of other classes (Supplier and Customer).

Composition vs inheritance

Limitations of inheritance

- Can create **deep and rigid class hierarchies** that are hard to modify or extend.
- **Tight coupling** between child and parent classes—changes in one can break the other.
- Subclasses may inherit **unwanted or irrelevant behavior**.

Composition vs inheritance

Advantages of composition over inheritance

- Classes are built from smaller components (*"has-a"* relationships), increasing **flexibility and modularity**.
- Behaviors can be mixed, matched, or replaced **without changing class hierarchies**.
- **Reduces coupling** and makes code easier to maintain, extend, and reuse.
- **Guideline:** Favor composition over inheritance for maximum flexibility and maintainability.
- **Good uses of inheritance:** Use inheritance for clear "is-a" relationships.

Encapsulation, abstraction and composition in practice

Exercise: Implementing the `ProblemData` class

- Implement a class `ProblemData` to manage suppliers, customers, and costs between them.
- The class should:
  - Store information about all suppliers and customers, *each identified by a unique name*.
  - Manage the costs associated with shipping from each supplier to each customer.
  - Allow initialization with a list of cost objects.
  - Provide a way to add new cost objects after initialization.
  - Provide access to the complete lists of suppliers, customers, and costs.
  - Provide efficient retrieval of suppliers, customers, and costs by name.

Encapsulation, abstraction and composition in practice

The `ProblemData` class interface

```python
class ProblemData:
    def __init__(self, costs: list[Cost]): ...

    @property
    def suppliers(self) -> list[Supplier]: ...
    @property
    def customers(self) -> list[Customer]: ...
    @property
    def costs(self) -> list[Cost]: ...
    def add_cost(self, cost: Cost) -> None: ...
    def get_supplier(self,
                     supplier_name: str) -> Supplier: ...
    def get_customer(self,
                     customer_name: str) -> Customer: ...
    def get_cost(self, supplier_name: str,
                 customer_name: str) -> Cost: ...
```

Encapsulation, abstraction and composition in practice

Key design strengths of the `ProblemData` class

- Internal data is kept private, preventing external access. **(Encapsulation)**
- Provides a simple interface; users don't see or depend on internal details like dictionaries. **(Abstraction, Encapsulation)**
- Manages `Supplier`, `Customer`, and `Cost` objects, not raw data. **(Composition)**
- Centralized data logic avoids duplication and inconsistency. **(Encapsulation)**
- Implementation can change (e.g., new data structures, added validation) without breaking user code. **(Abstraction, Encapsulation)**

Static methods

- A **static method** is a function defined within a class that does not access or modify instance or class data.
- Use static methods for helper or utility functions that are logically related to the class, but do not need object state.
- **Syntax:**

```
class MyClass:
    @staticmethod
    def my_static_method(args):
        # method body
```

- **Note:** A static method does not receive `self` as an argument.
- Static methods can be called on the class itself:

```
MyClass.my_static_method()
```

Static methods

Example: get_dict_from_csv in ProblemData

```
class ProblemData:
    ...
    @staticmethod
    def get_dict_from_csv(file: str, index: str | list[str],
                          col: str):
        data_df = pd.read_csv(file)
        data_dict = data_df.set_index(index).to_dict()[col]
        return data_dict
```

- get_dict_from_csv is a static utility method for reading and transforming CSV data into dictionaries.
- It does not use any attributes of ProblemData or its instances.
- Organizing such helpers within the class keeps code tidy and logical.
- It can be called directly on the class, without creating a ProblemData object:

```
supply_dict = ProblemData.get_dict_from_csv(
    supply_path, "supplier", "supply")
```

Class methods

- A **class method** is a function defined within a class that receives the class (`cls`) as its first argument.
- Use class methods for operations that need to **access or modify class-level data** or that provide **alternative ways to construct instances**.
- **Syntax:**

```
class MyClass:
    @classmethod
    def my_class_method(cls, args):
        # method body
```

- **Note:** A class method receives `cls` (the class itself) as its first argument, not `self`.
- Class methods can be called on the class itself:

```
MyClass.my_class_method()
```

Class methods

Example: from_csvs in ProblemData

```
@classmethod
def from_csvs(cls, supply_path: str, demand_path: str,
              cost_path: str) -> "ProblemData":
    supply_dict = cls.get_dict_from_csv(
        supply_path, "supplier", "supply")
    ...
    for (sup_name, cust_name), value in costs_dict.items():
            ...
        costs.append(Cost(supplier, customer, cost_val))
    return cls(costs)
```

- from_csvs is a class method that provides an alternative way to construct a ProblemData object from CSV files:

```
problem_data = ProblemData.from_csvs(
    supply_path, demand_path, cost_path)
```

- It uses cls to refer to the class and can call other class methods or static methods, like get_dict_from_csv.

Polymorphism

- **Polymorphism** is the ability of objects of different classes to respond to the same method call in ways specific to their own class.
- This allows you to call the same method on different objects and get behavior appropriate to their class.

```
entities = [Supplier("A", 10), Customer("B", 20)]
for entity in entities:
    entity.print()  # Calls Supplier.print or Customer.print
```

- This enables flexible and uniform code that works with different object types.

Abstract classes

- An **abstract class** defines a common interface for its subclasses, but is not meant to be instantiated directly.
- Abstract classes often include one or more methods that must be implemented by subclasses.
- They help enforce a **contract** for subclasses, ensuring consistent method names and usage.
- Used to represent **generic concepts**, with concrete details filled in by subclasses.

Abstract classes

Syntax

```python
class AbstractClass:
    def do_something(self):
        pass  # No implementation

class ConcreteClass(AbstractClass):
    def do_something(self):
        # Concrete implementation
        ...

# Only the concrete class can be instantiated
my_instance = ConcreteClass()
```

Abstract classes

Example: Abstract `Solver` class

```python
class Solver:
    def solve(self, data: ProblemData) -> FlowDict:
        pass

class GreedySolver(Solver):
    def solve(self, data: ProblemData) -> FlowDict:
        # Implementation of greedy heuristic
        ...
```

- `Solver` defines the **interface** (the `solve()` method), but does not implement it.
- `GreedySolver` **implements** the `solve()` method with specific logic.
- This enforces a common interface for all solvers.

Abstract base classes

The ABC module: Defining an abstract base class

- In Python, the abc module provides tools for defining abstract base classes.
- Abstract base classes declare methods that must be implemented by any subclass.

```
from abc import ABC, abstractmethod

class MyBase(ABC):
    @abstractmethod
    def do_something(self):
        pass
```

Abstract base classes

The ABC module: Defining subclasses

- Subclasses must implement all methods marked with `@abstractmethod`.
- If a subclass does not implement all abstract methods, it cannot be instantiated and Python will raise an error.

```
class MyConcrete(MyBase):
    def do_something(self):
        print("Implemented!")

class BrokenConcrete(MyBase):
    pass

obj = MyConcrete() # Works
obj2 = MyBase()     # Error: Can't instantiate abstract class
obj3 = BrokenConcrete() # Error: do_something not implemented
```

Abstract base classes

Example: Solver as an abstract base class

```
from abc import ABC, abstractmethod

class Solver(ABC):
    @abstractmethod
    def solve(self, data: ProblemData) -> FlowDict:
        pass

class GreedySolver(Solver):
    def solve(self, data: ProblemData) -> FlowDict:
        # Implementation
        ...
```

- `Solver` defines a **contract**: subclasses must implement `solve()`.
- Subclasses like `GreedySolver` must provide their own **implementation**.
- Attempting to instantiate `Solver` directly will raise an error.

Duck typing

- **Duck typing**: "If it walks like a duck and quacks like a duck, it's a duck."
- In Python, whether an object is suitable for a task depends on the methods and attributes it provides, not on its inheritance or type.
- An object can be used wherever a certain method is expected, as long as it implements that method; **inheriting from a specific class is not required**.
- This is a form of dynamic (runtime) checking.

Duck typing

Example

- Any object with a `solve()` method can be used as a solver, even if it does not inherit from `Solver`.

```
class CustomSolver:
    def solve(self, data):
        # Custom implementation
        ...
class TransportationProblem:
    def __init__(self, ..., solver):
        self.__solver = solver
        ...
    def solve(self):
        flows = self.__solver.solve(self.__data)
        ...

tp = TransportationProblem(..., CustomSolver())
tp.solve()
```

- `CustomSolver` works because it has the required `solve()` method, regardless of its type.

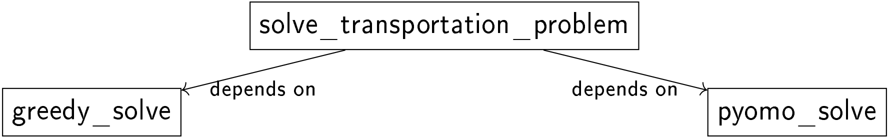
Dependency inversion principle

Imperative Approach: Direct dependency

- In an imperative design, a specific solver is called directly:

```python
def solve_transportation_problem(data, method) -> FlowDict:
        ...
    if method == "greedy":
        flows = greedy_solve(data)
    elif method == "pyomo":
        flows = pyomo_solve(data)
    ...
    return flows

flows = solve_transportation_problem(problem_data, "greedy")
```

- **Dependency diagram:**



Dependency inversion principle

Dependency Inversion: Abstract solver

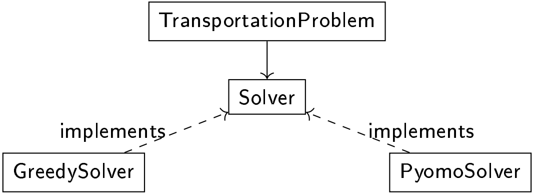
- The main logic (`TransportationProblem`) now depends only on the abstract `Solver` interface. Concrete solvers are injected from outside.

```
class TransportationProblem:
    def __init__(self, supply_file: str, demand_file: str,
                 costs_file: str, solver: Solver):
        self.__data = ProblemData.from_csvs(
            supply_file, demand_file, costs_file)
        self.__solver = solver

def solve(self) -> FlowDict:
    ...
    flows = self.__solver.solve(self.__data)
    ...

tp1 = TransportationProblem(..., GreedySolver())
flows1 = tp1.solve()
tp2 = TransportationProblem(..., PyomoSolver("cbc"))
flows2 = tp2.solve()
```

Dependency inversion principle

Dependency Inversion: Decoupled design

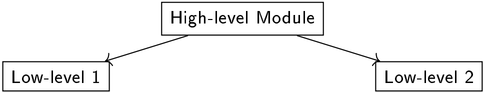
- The main logic is decoupled from specific solver implementations.
- Any solver can be used without changing `TransportationProblem`.
- **Dependency diagram:**



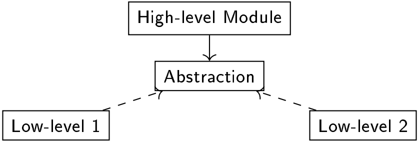
Dependency inversion principle

Direct dependency vs. dependency inversion

- **Direct dependency:** High-level code depends on implementations. Changes ripple through the codebase.



- **Dependency inversion:** High-level and low-level code depend on abstractions. New implementations can be added or swapped easily.



Magic methods

- **Magic methods** (also called "dunder" methods) are special methods with double underscores, like `__str__`, `__repr__`, `__getitem__`, etc.
- They allow classes to interact naturally with Python syntax and built-in functions:
  - `str(obj)`, `repr(obj)`
  - Indexing: `obj[key]`
  - Iteration: `for x in obj`
  - Length: `len(obj)`
  - Comparison: `obj1 == obj2`
- Magic methods make custom classes behave more like built-in types.

Magic methods

String representation: `__str__`, `__repr__`

- `__str__`: Defines the user-friendly string representation, used by `print()`.
- `__repr__`: Defines the official/debug representation, shown when inspecting objects in interactive Python sessions (e.g., terminal, Jupyter notebook).
- **Example:**

```python
class Supplier:
    ...
    def __repr__(self):
        return f"Supplier({self.name!r}, {self.supply!r})"

    def __str__(self):
        return f"Supplier {self.name}: {self.supply} units"

supplier = Supplier("S1", 100)
print(supplier)        # Output: Supplier S1: 100 units
supplier               # Output: Supplier('S1', 100)
```

Magic methods
Operator overloading

- Python lets you redefine how operators work for your own classes.
- This feature is called **operator overloading**.
- It is done using special **magic methods**. For example:
  - + : `__add__`
  - - : `__sub__`
  - * : `__mul__`
  - == : `__eq__`
  - < : `__lt__`
  - > : `__gt__`
- By defining these methods, you control how your objects behave with Python operators.

Magic methods

Example: overloading + with `__add__`

- By defining the `__add__` method, we can combine two Supplier objects using +:

```python
class Supplier:
    ...
    def __add__(self, other: "Supplier") -> "Supplier":
        # Combine names and supplies of two suppliers
        return Supplier(self.name + "|" + other.name,
                        self.supply + other.supply)

s1 = Supplier("S1", 100)
s2 = Supplier("S2", 200)
s3 = s1 + s2  # Calls s1.__add__(s2)
print(s3.name)    # Output: S1|S2
print(s3.supply)  # Output: 300
```

Magic methods

Sorting: `__eq__` and `__lt__`

- `__eq__` and `__lt__` let Python compare and sort objects.
- If you define them, you can sort objects directly with `sorted()`.
- **Example:**

```python
class Cost:
    ...
    def __eq__(self, other: 'Cost') -> bool:
        return self.value == other.value

    def __lt__(self, other: 'Cost') -> bool:
        return self.value < other.value

# Now you can sort a list of costs directly:
sorted_costs = sorted(data.costs)
```

- No need for a `key` function—Python uses these magic methods automatically.

Magic methods

Custom Indexing: `__getitem__`

- `__getitem__` lets you use square bracket access on your objects, just like with dictionaries.
- **Example:**

```python
class ProblemData:
    ...
    def __getitem__(self, key):
        # key is a tuple: (supplier_name, customer_name)
        return self.get_cost(key[0], key[1])

...
data = ProblemData(costs)
cost = data["S1", "C2"]  # returns Cost object from S1 to C2
```

- This makes your class more intuitive and “Pythonic”.