Abstract Base Classes

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Jake Roach
Data Engineer

datacamp

Abstract base classes

Abstract base classes create a **blueprint** for classes by defining **abstract methods that must be implemented** by all children

- Ensures common behavior for a group of classes
- `@abstractmethod` decorator
- Meant to be inherited, not instantiated
- Not all methods in abstract base class must be abstract (concrete methods)

```
class School:
    # Do things such as enroll
    # and add a course
```

```
class MiddleSchool:
    # Play an instrument, join a club
    # but must enroll and add courses
```

```
class HighSchool:
    # Play a varsity sport, apply
    # for college, but must enroll
    # and add courses
```

Creating an abstract base class

```
from abc import ABC, abstractmethod

# Create an abstract base class that inherits
# from ABC
class School(ABC):
    @abstractmethod
    def enroll(self):
        # This method must be implemented
        # in classes that inherit from it
        pass

    # Concrete methods are inherited
    def graduate(self):
        print("Congrats on graduating!")
```

- Inherit from the `ABC` class in the `abc` module
- `pass`
- Any class that inherits `School` must implement the `enroll()` method
- Can also implement concrete methods

Implementing abstract base classes

```
class HighSchool(School):
    def enroll(self):
        print("Welcome to high school!")

# Create an instance of HighSchool
high_school = HighSchool()
high_school.enroll()
```

```
Welcome to high school!
```

```
high_school.graduate()
```

```
Congrats on graduating!
```

- `HighSchool` **must** define an `enroll()` method
- `TypeError` if `enroll()` is not defined
- `HighSchool` inherits the `graduate()` method

Multiple abstract methods

```
class School(ABC):
    @abstractmethod
    def enroll(self):
        pass

    @abstractmethod
    def add_course(self, course_name):
        pass

    def graduate(self):
        print("Congrats on graduating!")
```

```
class HighSchool(School):
    def __init__(self):
        self.courses = []

    # Implementing abstract method
    def enroll(self):
        print("Welcome to high school!")

    # Implementing abstract method
    def add_course(self, course_name):
        self.courses.append(course_name)
        print(f"You enrolled in {course_name}")
```

- Two abstract methods, `enroll()` and `add_course()`

Let's practice!

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Interfaces

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

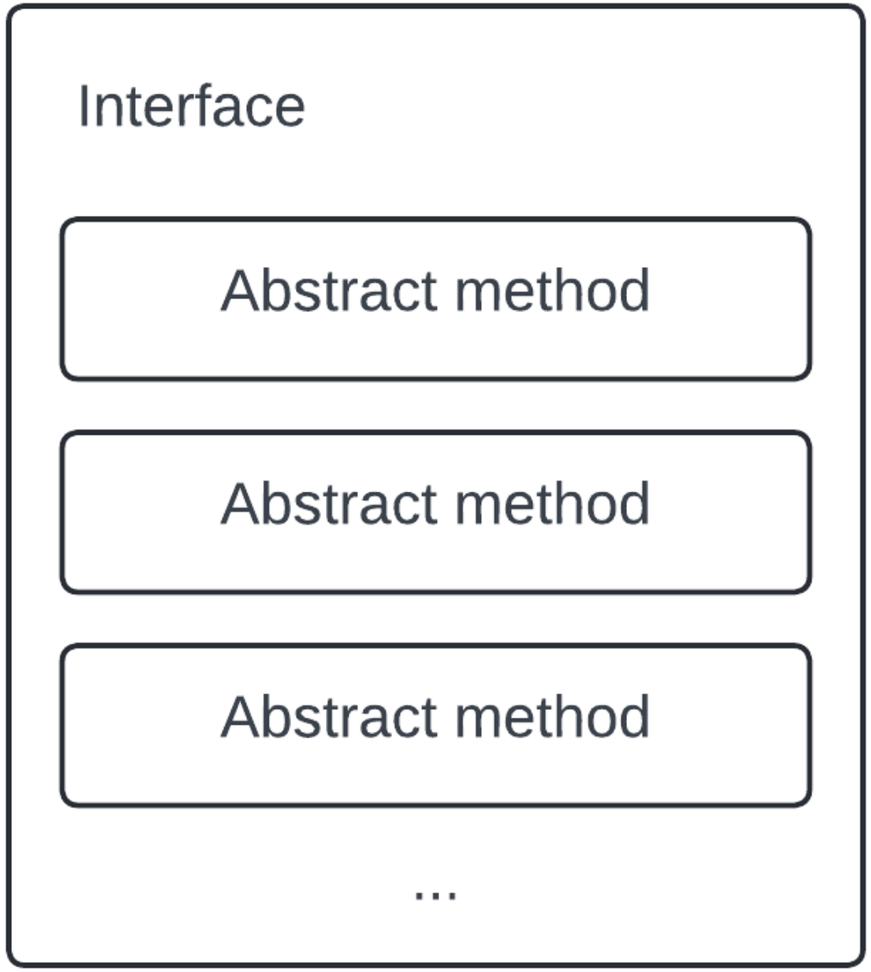
Jake Roach
Data Engineer

datacamp

Interfaces

A special kind of class **made up of only abstract methods** that create a *contract* with the classes that implement the interface

- Must define abstract methods
- Skeleton of method definitions
- Parameters do not need to match
- Formal or informal



Let's start with informal interfaces!

Informal interfaces with duck typing

```
class Course:
    def assign_homework(self, assignment_number, due_date):
        # Typically, the pass keyword will be used when creating an interface
        pass

    def grade_assignment(self, assignment_number):
        pass
```

- Duck typing
- `assign_homework()` and `grade_assignment()` abstract methods

Informal interfaces with duck typing

```
class ProgrammingCourse:
    def __init__(self, course_name):
        self.course_name = course_name

    def assign_homework(self, due_date):
        # Some implementation of assign_homework here...

    def grade_assignment(self, assignment_number):
        # Some implementation of grade_assignment here...
```

Formal interfaces with ABC, @abstractmethod

```
from abc import ABC, abstractmethod

class Course(ABC):
    @abstractmethod
    def assign_homework(self, assignment_number, due_date):
        pass

    @abstractmethod
    def grade_assignment(self, assignment_number):
        pass
```

How are formal interfaces different than abstract base classes?

Interfaces vs. Abstract Base Classes

Interfaces

- Define a "contract" with classes that implement it
- *"Must-do"*
- Formal and informal
- Only contain abstract methods (no concrete methods)
- Nearly identical classes

Abstract base classes

- Create a "blueprint" for classes with common characteristics
- *"Is-a"*
- Typically only formal
- Mix of abstract and concrete methods
- Classes that look and feel similar

Implementing a formal interface

```
class ProgrammingCourse(Course):
    def __init__(self, course_name):
        self.course_name = course_name

    def assign_homework(self, due_date):
        # Some implementation of assign_homework here...

    def grade_assignment(self, assignment_number):
        # Some implementation of grade_assignment here...

intermediate_oop = ProgrammingCourse("Intermediate Object-Oriented Programming")
```

Failing to implement a formal interface

```
class ProgrammingCourse(Course):
    def __init__(self, course_name):
        self.course_name = course_name

    def assign_homework(self, due_date):
        # Some implementation of assign_homework here...

intermediate_oop = ProgrammingCourse("Intermediate Object-Oriented Programming")
```

```
 ...
    intermediate_oop = ProgrammingCourse("Intermediate Object-Oriented Programming")
TypeError: Can't instantiate abstract class ProgrammingCourse with abstract method
    grade_assignment
```

Let's practice!

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Factory methods

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Jake Roach
Data Engineer

datacamp

Conditional logic in a method

```
class Student:
    # Conditional logic with similar behavior for all conditions
    def explore_topic(self, resource_type, topic):
        if resource_type == "Textbook":
            print(f"Referencing {topic} using a textbook")
            self.reference_textbook(topic)

        elif resource_type == "Blog":
            print(f"Make sure to validate information provided in blogs!")
            self.reference_blog(topic)

        elif resource_type == "Video":
            self.reference_video(topic)
```

Factory methods

Design pattern that uses factory methods to create objects that are used in another method

- Return objects that implement an interface
- Reduce complexity in a method
- Reusable, modular
- `_` to denote a factory method

First, let's take a look at products and concrete products!

```
...

  def _get_resource(self, resource_type):
    if resource_type == "Textbook":
      # Textbook, Blog and Video are
      # all resources
      return Textbook()

    elif resource_type == "Blog":
      return Blog()

    elif resource_type == "Video":
      return Video()
```

Creating a Resource interface

```
class Resource(ABC):
    @abstractmethod
    def reference(self, topic):
        pass
```

```
class Textbook(Resource):
    def __init__(self):
        self.index = {"Object-oriented Programming": ["Inheritance", ...]}

    def reference(self, topic):
        print(f"Referencing {topic} using a textbook")
        return self.index.get(topic)

# Something similar for Blog, Video
```

Reworking explore_topic()

```
class Student:

    def explore_topic(self, resource_type, topic):
        if resource_type == "Textbook":
            texbook = Textbook()  # Creating an instance of class implementing Resource
            texbook.reference(topic)  # Call the reference() method

        elif resource_type == "Blog":
            blog = Blog()
            blog.reference(topic)

        elif resource_type == "Video":
            video = Video()
            video.reference(topic)
```

Creating a factory method

```
class Student:
    # Factory method to return Resource
    def _get_resource(self, resource_type):
        if resource_type == "Textbook":
            return Textbook()
        elif resource_type == "Blog":
            return Blog()
        elif resource_type == "Video":
            return Video()

    def explore_topic(self, resource_type, topic):
        resource = self._get_resource(resource_type)  # Retrieve the resource
        return resource.reference(topic)
```

Using factory methods

```
# Create a Student object, then have it reference a textbook
lester = Student()
lester.explore_topic("Textbook", "Object-oriented Programming")

# Each to switch to another resource
lester.explore_topic("Video", "Object-oriented Programming")
```

```
Referencing Object-oriented Programming using a textbook
["Inheritance", "Constructors", "Class-level methods"]

Video are helpful for visual or audatory learners
["Classes", "Methods", "Attributes", "self"]
```

Let's practice!

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Congratulations!

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Jake Roach
Data Engineer

datacamp

Intermediate Object-Oriented Programming with Python

Multiple inheritance

Operator overloading

Type hints

Descriptors

`__setattr__()`

`__getattr__()`

Custom iterators

Abstract base classes

Interfaces

Factory methods

Intermediate Object-Oriented Programming with Python

Multiple inheritance

Operator overloading

Type hints

Descriptors

`__setattr__()`

`__getattr__()`

Custom iterators

Abstract base classes

Interfaces

Factory methods

Intermediate Object-Oriented Programming with Python

Multiple inheritance

Operator overloading

Type hints

Descriptors

`__setattr__()`

`__getattr__()`

Custom iterators

Abstract base classes

Interfaces

Factory methods

Thank you!

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON