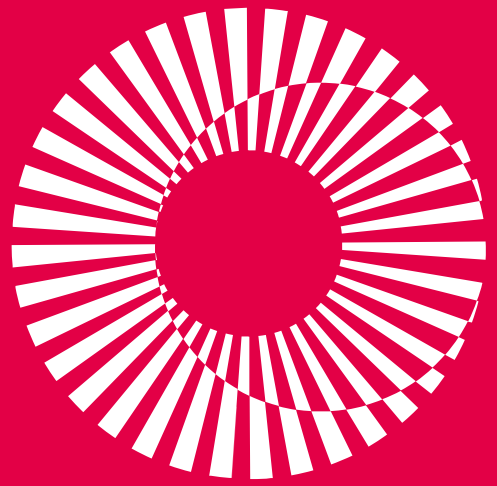


UCLL
HOGESCHOOL

Abstract Methods – Abstract Properties
Programming 2

Janne Gilis – Lies Bollens – Ruben Naudts – Bart Thumas – Rob Coekaerts

PBA Toegepaste Informatica – PBA Applied Computer Science
2025 – 2026

Agenda

Abstract Methods

Abstract Properties

Revision

Agenda

Abstract Methods

Abstract Properties

Revision

Abstract classes

```
from abc import ABC

class Shape(ABC):
    ...
```

- Abstract classes are classes that cannot be instantiated and that contain abstract methods
- Use of the ABC (Abstract Base Class) module in Python

Abstract methods

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass
```

- Abstract methods are methods declared in an abstract class, without providing an implementation
- Use of the @abstractmethod decorator to declare abstract methods

Abstract class vs. concrete class

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

obj = Shape() # Results in Error
obj.calculate_area() # Results in Error
```

- Cannot be instantiated directly. It exists to be subclassed
- Its main purpose is to serve as a blueprint for other classes

Abstract class vs. concrete class

```
class Rectangle():
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

rectangle = Rectangle(width=5, height=10)
print("Rectangle Area:", rectangle.calculate_area())
```

- Can be instantiated directly and used to create objects
- Inheritance: provides implementations for all the abstract methods defined in its abstract parent class

Abstract class inheritance

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass
```

```python
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius * self.radius

circle = Circle(radius=7)
print("Circle Area:", circle.calculate_area())
```

- Why? BLUEPRINT!!!

Abstract methods and polymorphism

- Abstract methods allow dynamic binding, meaning the right method is chosen at runtime based on the actual object type.

- Each subclass can implement the abstract method in its own way but must follow the rules set by the abstract class, we mean that the subclass has to implement the abstract methods declared in the abstract class.

- This enables runtime polymorphism, where different subclasses can have different method implementations, giving the code more flexibility and extensibility.

Agenda

Abstract Methods

Abstract Properties

Revision

Abstract Properties

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @property
    @abstractmethod
    def color(self):
        pass
```

- Abstract properties are attributes that are declared but not implemented in an abstract class
- Order matters, you have to use @property above @abstractmethod

Getter Method for abstract property

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @property
    @abstractmethod
    def color(self):
        pass
```

- Read-only abstract properties by omitting the setter method
- Order matters, you have to use @property above @abstractmethod

Setter Method for abstract property

```
from abc import ABC, abstractmethod

class Shape(ABC):
    ...

    @color.setter
    @abstractmethod
    def color(self, value):
        pass
```

- Order matters, you have to use @attribute.setter above @abstractmethod

Abstract property in concrete subclass

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @property
    @abstractmethod
    def color(self):
        pass

    @color.setter
    @abstractmethod
    def color(self, value):
        pass

    @abstractmethod
    def calculate_area(self):
        pass
```

```python
class Rectangle(Shape):
    def __init__(self, width, height, color):
        self.width = width
        self.height = height
        self.color = color

    @property
    def color(self):
        return self._color

    @color.setter
    def color(self, value):
        if not isinstance(value, str):
            raise ValueError("Color must be a string")
        self._color = value

    def calculate_area(self):
        return self.width * self.height
```

```python
rectangle = Rectangle(width=5, height=10, color="Blue")

print("Rectangle Area:", rectangle.calculate_area())
print("Rectangle Color:", rectangle.color)

rectangle.color = 42 # Results in error due to color validation
```

- Validation is done in the implementation of the abstract setter property

Use Cases for abstract properties

- Enforcing consistency: Ensure that all subclasses have the same property with the same name.
- Providing flexibility: Let each subclass define its own logic or behavior for the property.
- Data encapsulation: When you want to control access to an attribute via a property but require subclasses to define its behavior.
- ....

Agenda

Abstract Methods

Abstract Properties

Revision

Abstract properties vs. abstract methods

Properties

- Defining attributes without providing the implementation for their getter and setter methods
- Use the @property decorator for the getter and @attribute.setter for the setter, on top of the @abstractmethod decorator
- Want to ensure that all subclasses provide a specific set of attributes with getter and setter methods

Methods

- Defining a method signature without providing any implementation
- Use the @abstractmethod decorator directly on the method
- Want to ensure that all subclasses provide a specific method implementation

Exercise

Try the following exercises

01-OO

- 07-abstract-classes

